PROGRAMAÇÃO CONCORRENTE BASEADA EM ACORDES PARA PLATAFORMA JAVA

SÉRGIO VALE E PACE

PROGRAMAÇÃO CONCORRENTE BASEADA EM ACORDES PARA PLATAFORMA JAVA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: ROBERTO DA SILVA BIGONHA

Belo Horizonte

Julho de 2009

Pace, Sérgio Vale e
P115p Programação Concorrente Baseada em Acordes para Plataforma Java / Sérgio Vale e Pace. — Belo Horizonte, 2009
xxvi, 141 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Roberto da Silva Bigonha

1. Computação - Teses. 2. Programação concorrente - Teses. 3. Linguagens de programação de computador - Teses. 4. JAVA (Linguagem de programação de computador) - Teses. I. Título.

CDU 519.6*33

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Programação concorrente baseada em acordes para plataforma Java

SÉRGIO VALE E PACE

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. ROBERTO DA SILVA BIGONHA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE
Departamento de Ciência da Computação - PUC - MG

PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

PROF. OSVALDO SÉRGIO FARHAT DE CARVALHO
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 26 de junho de 2009.

Dedico este trabalho a pessoa que é, e sempre foi, um exemplo de fé inabalável, espírito inquebrável e amor incondicional. Minha avó Áurea.

Agradecimentos

Agradeço ao professor Roberto Bigonha pela orientação, paciência e dedicação, que foram fundamentais para a realização desse trabalho.

Agradeço também à professora Mariza Bigonha, ao professor Osvaldo Carvalho e ao professor Marco Túlio Valente pelos muitos comentários, revisões e sugestões.

Agradeço à minha mãe Áurea e ao meu pai Carlos, que sempre me proporcionaram quase tudo o que eu podia querer e absolutamente tudo o que eu precisei pra me tornar quem eu sou hoje.

Agradeço a meu grande amigo e irmão Ronaldo por todas as lições que me ensinou, mesmo sem querer.

Agradeço a todos os meus amigos e colegas da AAIS (ATAN) pelo companheirismo, apoio e amizade, sem os quais esse trabalho não seria o mesmo.

Por fim, mas não menos importante, agradeço à minha Aline. Pelas madrugadas de trabalho, pela atenção dividida, pelos planos postergados, mas principalmente por ser a metade da minha laranja e o norte na minha bússola.

"Poor workers blame their tools.
Good workers build better tools.
The best workers get their tools
to do the work for them. "
(The Book of Cataclysm - Syndicate Wars)

Resumo

Os principais mecanismos usados para expressar paralelismo e concorrência disponíveis nas principais linguagens de programação modernas são construções de baixo nível de abstração, inadequadas ao desenvolvimento de sistemas concorrentes de larga escala. Isso faz com que a tarefa de projetar, analisar, implementar, testar e depurar sistemas concorrentes seja bastante árdua.

O acorde é uma construção de alto nível, baseada no cálculo de processos *join-calculus*, que permite expressar os relacionamentos de concorrência declarativamente, possibilitando coordenar as interações entre fluxos de execução paralelos de maneira implícita.

Propõe-se uma biblioteca para programação concorrente baseada em acordes na plataforma Java. Por meio dessa biblioteca busca-se permitir que sistemas paralelos e concorrentes sejam desenvolvidos em um maior nível de abstração de forma mais simples, efetiva e menos propensa a erros. É utilizada uma implementação baseada em anotações e instrumentação dinâmica de *bytecodes* que permite adaptar a sintaxe e a semântica da linguagem, propiciando expressar os acordes de maneira natural sem comprometer o ambiente padrão Java de desenvolvimento e execução.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Capítulo 1

Introdução

Programação paralela e concorrente é um componente importante para sistemas computacionais modernos. Com o advento da "revolução *Multi-Core*" [Sutter, 2005], essa modalidade de programação passa a desempenhar um papel ainda mais crucial na evolução da computação de forma geral. Entretanto, os mecanismos usados para expressar paralelismo e concorrência disponíveis nas principais linguagens de programação modernas ainda são construções de baixo nível, baseadas em conceitos que evoluíram muito pouco nos últimos 30 anos, inadequadas ao desenvolvimento de sistemas concorrentes de larga escala [Itzstein, 2005].

Desde o início de 2003, devido a uma série de fatores técnicos e limites físicos fundamentais, os principais fabricantes de processadores passaram a utilizar, em larga escala, as arquiteturas *multi-core*. Nesse tipo de arquitetura, os processadores dispõem de dois ou mais núcleos de processamento independentes (*cores*). Busca-se assim, aumentar a capacidade de processamento pela execução simultânea de duas ou mais instruções. Isso é diferente da estratégia adotada até então de aumentar a capacidade de processamento pela diminuição do tempo de execução individual de cada instrução. [Sutter, 2005].

As principais linguagens de programação modernas são fundamentalmente sequenciais. A *thread* é hoje a abstração mais usual e amplamente utilizada em linguagens de programação e sistemas operacionais para expressar concorrência e paralelismo. De acordo com Lee [2006] muitas das arquiteturas paralelas de propósito geral em uso hoje são uma implementação direta em *hardware* da abstração de *thread*.

Normalmente, *threads* são implementadas como adições periféricas à sintaxe da linguagem (Java, C#) ou como bibliotecas externas (C++ [Butenhof, 1997])[Lee, 2006], modeladas diretamente a partir das primitivas do sistema operacional [Benton et al., 2004]. A programação baseada em *threads* é complexa e propensa a erros graves e difíceis de detectar, uma vez que as *threads* introduzem não-determinismo implícito e

irrestrito ao processo. Isso faz com que a tarefa de projetar, analisar, implementar, testar e debugar sistemas concorrentes seja bastante árdua [Benton et al., 2004].

Muito embora a programação baseada em *threads* seja a técnica dominante para o desenvolvimento de sistemas paralelos e concorrentes nas principais linguagens de programação modernas, novas técnicas têm sido propostas. Dentre as alternativas temos a programação baseada em acordes.

O acorde é uma construção de alto nível, baseada no cálculo de processos *join-calculus*, que permite expressar os relacionamentos de concorrência declarativamente, possibilitando que as interações entre fluxos de execução paralelos sejam coordenados de maneira implícita [Petrounias et al., 2008].

Os acordes se ajustam muito bem às linguagens de programação imperativas orientadas a objetos. Dentre essas linguagens, uma das mais usadas atualmente é a linguagem Java [Gosling et al., 2005], que semanticamente se mostra bastante adequada à programação baseada em acordes. Contudo, são necessárias algumas adições sintáticas à linguagem Java para que os acordes possam ser usados confortavelmente.

Itzstein [2005] e Wood & Eisenbach [2004] implementam acordes em Java por meio de compiladores customizados. Essa abordagem restringe a aplicação prática dessas soluções, principalmente em ambientes de produção. Com isso o acesso à programação baseada em acordes fica limitada a um grupo restrito de desenvolvedores Java.

Propõe-se nesse trabalho a criação de uma biblioteca de classes para programação baseada em acordes na linguagem Java. Essa biblioteca é baseada em anotações e manipulações de *bytecode*. Essa abordagem possibilitará a utilização de acordes com ferramental Java padrão. A essa biblioteca dar-se-á o nome JChords.

1.1 *Threads*

Threads são, essencialmente, processos sequenciais, que compartilham recursos entre si. Sintaticamente, acomodar o conceito de *threads* em uma linguagem sequencial não requer muito esforço, seja por meio de primitivas da própria linguagem ou via bibliotecas externas. Talvez a isso se deva o sucesso da *thread* como abstração de concorrência [Lee, 2006].

Contudo, semanticamente, as *threads* representam um desvio significativo dos conceitos fundamentais nos quais se baseiam a maior parte das linguagens de programação de propósito geral. A maioria das linguagens de programação busca ser compreensível, predizível e determinística. No paradigma sequencial, essas propriedades são obtidas enfatizando sequências de ações determinísticas, formando composições também determinísticas [Lee, 2006].

Seja B_n uma sequência longa, porém finita, de dígitos binários que representam o estado completo de um dado sistema computacional em um dado instante n. Essa sequência de dígitos engloba de maneira completa toda e qualquer informação de estado desse sistema, incluindo células de memória, registradores ou *flags* do processador, arquivos armazenados em disco, entre outros. Ou seja, o estado interno de cada subsistema pode ser mapeado em uma subsequência de B_n de modo que qualquer parte do sistema que seja mutável no tempo equivale a uma sequência de *bits* em B_n

Seja então P_1 o processador desse computador, ou seja, o elemento ativo desse sistema. Nesse contexto, P_1 é o único agente de mudança e, por consequência, o único responsável pela transição de B_n para B_{n+1}. Dessa forma podemos dizer que B_{n+1} é o produto da ação de P_1 sobre B_n, ou seja $B_{n+1} = P_1(B_n)$ para um dado B_n qualquer:

$$B_0 \xrightarrow{P_1} B_1 \xrightarrow{P_1} B_2 \xrightarrow{P_1} \cdots \xrightarrow{P_1} B_n \tag{1.1}$$

Podemos observar que nesse cenário existe um único caminho entre um estado B_n qualquer e qualquer outro estado B_m, tal que $m > n$. A partir de um estado qualquer, sabemos *a priori* quais serão os estados futuros e quantos passos são necessários para alcançá-los, ou seja, temos previsibilidade e determinismo.

Por sua vez, sistemas baseados em *threads* observam-se não-determinísticos por definição. Uma *thread* pode ser vista, dentro do modelo proposto acima, como um segundo processador P_2 atuando paralelamente em B_n. Dois processos independentes P_1 e P_2, atuando simultaneamente sobre um estado B_n irão produzir um estado B_{n+1} dentre um número potencialmente ilimitado de possibilidades. Para enumerar alguns temos:

- As ações de P_1 sobrescrevem completamente as ações de P_2, logo $B_{n+1} = P_1(B_n)$
- Todas as ações de P_1 sobre B_n ocorrem antes de P_2, logo $B_{n+1} = P_2(P_1(B_n))$
- As ações de P_1 ocorrem simultaneamente as de P_2, logo $B_{n+1} = P_1|P_2(B_n)$

Dessa forma a transição de B_n para B_{n+1} passa a ser não-determinística e consideravelmente menos previsível:

$$B_0 \xrightarrow{P_1|P_2} \begin{cases} P_1(B_0) \\ P_1(P_2(B_0)) \\ \cdots \\ P_1|P_2 \\ \cdots \\ P_2(P_1(B_0)) \\ P_2(B_0) \end{cases}$$

Dessa forma, a introdução do conceito de *threads* em uma linguagem sequencial compromete o determinismo desta e elimina as características mais significativas do paradigma sequencial.

Consequentemente, sistemas *multithread* requerem um nível de esforço e complexidade elevados em sua análise, desenvolvimento e manutenção. Isso porque se torna necessário construir e manter um conjunto significativo de ferramentas, técnicas, padrões e construções que permitam ao desenvolvedor limitar o não-determinismo introduzido pelas *threads*.

Em essência, todas as construções de sincronização presentes hoje nas linguagens de programação de propósito geral: semáforos, monitores, exclusão mútua, *locks*, dentre outras; servem principalmente para permitir ao desenvolvedor restringir os caminhos indesejáveis na computação *multithread* [Lee, 2006].

Contudo, para qualquer sistema paralelo não-trivial o número de interações indesejáveis entre *threads* é potencialmente infinito, e assim, mesmo dispondo dos meios para controlar e sincronizar *threads* essa ainda é uma tarefa árdua e altamente sujeita a erros, que normalmente são muito difíceis de detectar.

As *threads* são um importante bloco construtivo para a programação paralela e concorrente em linguagens de programação imperativas. Porém, como vimos nessa seção, são construções de baixo nível de abstração que produzem efeitos colaterais bastante significativos, que normalmente não são imediatamente claros. Segundo Lee [2006], *threads* não deveriam ser usadas explicitamente no desenvolvimento de sistemas concorrentes, mas sim como base para construções com maior nível de abstração. Uma analogia válida são os comandos de desvio de execução (*goto*) que, embora presentes em algum nível de abstração, não são explicitamente usados ou suportados na maior parte das linguagens de programação modernas.

1.2 Programação Concorrente Baseada em Acordes

A necessidade de abstrações adequadas a modelagem e orquestração de sistemas concorrentes levou a criação de diversas álgebras de processos dedicados a modelagem formal de sistemas concorrentes. *Join-Calculus* é um exemplo desse tipo de álgebra. O *join-calculus*, proposto por Fournet & Gonthier [2000], foi desenvolvido para atender às necessidades de modelamento de sistemas distribuídos [Benton et al., 2004].

Baseado no *join-calculus*, Benton et al. [2002] e Itzstein & Kearney [2002] propuseram uma construção chamada Acorde (*Chord* no inglês). O Acorde se propõe a ser uma construção de elevado grau de abstração para expressar situações complexas de concorrência e sincronização em linguagens imperativas baseadas no paradigma orientado por objetos [Wood & Eisenbach, 2004].

Nos primórdios da programação orientada a objetos, particularmente em Smalltalk, a separação conceitual entre mensagem e método era bastante bem definida. Mensagens são sinais parametrizados que podem ser enviados a um objeto. Métodos são conjuntos de ações que devem ser executadas em momentos específicos. Normalmente o recebimento de uma mensagem causa a execução de um método [Ingalls, 1981].

Esse relacionamento estreito entre mensagens e métodos fez com que a separação entre esses dois elementos fosse ficando indistinta, à medida que foram surgindo novas linguagens orientadas a objetos como C++, Java, C#, dentre outras. Em geral, nessas linguagens, o envio de uma mensagem corresponde a execução de exatamente um método. O método específico que será executado irá depender do objeto que recebe a mensagem, contudo somente um único método de um único objeto é executado em resposta ao recebimento de uma mensagem. Isso muitas vezes cria a ilusão de identidade entre esses dois elementos.

Em acordes, a distinção entre método e mensagem passa a ser relevante, pois a execução de um método pode requerer o recebimento de mais de uma mensagem. Sintaticamente, o acorde é construído por um conjunto de mensagens ligadas pelo operador *join* (representado aqui pelo caractere &) e o corpo de um método. Respectivamente a assinatura e o corpo do acorde como ilustrado na Listagem 1.1 [Drossopoulou et al., 2006].

```
1 Mensagem1(parâmetro1) & Mensagem2(parâmetro2) {}
```

Listagem 1.1. Acordes

As mensagens que compõem a assinatura do acorde, também chamadas de fragmentos do acorde, podem ser síncronas ou assíncronas. Um acorde possui, no máximo, uma única mensagem síncrona. Somente mensagens síncronas podem retornar valor. Por consequência, um acorde só pode retornar um único valor. Por sua vez, o corpo

do acorde tem acesso a todos os parâmetros de todas as mensagens que compõem a assinatura e, se necessário, deve prover o valor de retorno do acorde. [Drossopoulou et al., 2006].

O corpo de um acorde só é executado quando todas as mensagens da assinatura tiverem sido recebidas. As mensagens assíncronas não interrompem o fluxo de execução onde foram emitidas. As mensagens síncronas bloqueiam o fluxo de execução até que o corpo do acorde tenha sido executado [Benton et al., 2007].

```
1 public class Buffer {
2   public String get() & public async put(String s) {
3       return s;
4   }
5 }
```

Listagem 1.2. *Buffer* com acordes

No código de exemplo da Listagem 1.2, adaptado de Benton et al. [2007], temos uma classe `Buffer` que possui somente um acorde, que é composto pela mensagem síncrona `String get()`, pela mensagem assíncrona `put(String s)`, e por um método que retorna o parâmetro `s` de `put(String s)`.

O envio de uma mensagem `put(String s)` para uma instância de `Buffer` não retorna valor, não bloqueia o fluxo de execução e, por si só, não causa a execução do corpo do acorde, ou seja, `put(String s)` é condição necessária, mas não suficiente para causar a execução do corpo do acorde.

Em contrapartida, a emissão de uma mensagem `get()` para uma instância de `Buffer` irá retornar um valor do tipo `String` e irá bloquear o fluxo de execução do programa até que o corpo do acorde tenha condições de ser executado, porém, por si só, também não é capaz de causar essa execução [Benton et al., 2004].

```
1 Buffer buffer = new Buffer();
2 buffer.put(``A'');
3 buffer.put(``B'');
4
5 System.Console.WriteLine(buffer.get()); // imprime ``B''
6 System.Console.WriteLine(buffer.get()); // imprime ``A''
7 System.Console.WriteLine(buffer.get()); // bloqueia
```

Listagem 1.3. Uso do *buffer* com acordes

Como pode ser observado no exemplo da Listagem 1.2 e da Listagem 1.3, a assinatura de um acorde provê uma salvaguarda para a execução do corpo do mesmo,

promovendo uma declaração de condições necessárias para a execução deste [Petrounias et al., 2008].

Vale ressaltar também que a gestão de recursos compartilhados, no caso a *string* "s", deixa de ser uma responsabilidade do desenvolvedor e passa a ser responsabilidade da implementação dos acordes[Itzstein, 2005].

O exemplo dado, apesar de bastante simples, por si só nos permite solucionar de maneira efetiva vários problemas do tipo Produtor-Consumidor. O acorde é uma construção bastante poderosa que permite o modelamento de problemas mais complexos de concorrência de uma maneira efetiva e direta.

Threads introduzem um alto grau de não-determinismo nos sistemas paralelos. Para contornar esse problema foram criadas mecanismos para permitir ao desenvolvedor restringir os possíveis caminhos de execução de um sistema paralelo. Pela utilização de acordes, adotou-se a abordagem oposta, ao invés de tentar inibir o conjunto potencialmente infinito de computações indesejáveis, os acordes nos permitem expressar de maneira elegante e efetiva o conjunto potencialmente finito de computações desejáveis, introduzindo o não determinismo apenas em pontos específicos e controlados do sistema.

1.3 Soluções Existentes

Acordes tem sido foco de diversos projetos de pesquisa e, consequentemente, têm surgido diversas iniciativas de implementar acordes em novas ou existentes linguagens de programação orientadas a objetos. Essas iniciativas variam em propósito, indo desde de iniciativas voltadas para o desenvolvimento formal da teoria de acordes, como Drossopoulou et al. [2006] e Petrounias et al. [2008], até soluções dedicadas a aplicação imediata em ambiente produtivo como Russo [2006].

As primeiras implementações de acordes conhecidas foram feitas por Fournet et al. [2002], Benton et al. [2002] e Itzstein & Kearney [2002]. Esses trabalhos cooperativamente lançaram as bases semânticas e sintáticas para o desenvolvimento prático baseado em acordes. Dentre essas, Benton et al. [2002] e Itzstein & Kearney [2002] foram as primeira implementações baseadas em linguagens orientadas a objetos. Ambas foram construídas pela implementação de compiladores customizados para C# e Java, respectivamente.

No âmbito do desenvolvimento teórico, Drossopoulou et al. [2006] propôs uma nova linguagem inteiramente baseada em acordes. Essa iniciativa buscava um melhor entendimento dos acordes como construções de linguagem e procurou criar um conjunto mínimo de funcionalidades necessárias para obter uma linguagem imperativa orien-

tada a objetos com suporte a acordes. Posteriormente, em Petrounias et al. [2008], foi demonstrado que o suporte a variáveis membros (campos) de uma classe pode ser emulados por meio de acordes, sugerindo aplicações mais amplas para os mesmos.

A partir de Benton et al. [2002], o esforço para disponibilizar acordes em C# continuou em Benton [2003] e Benton et al. [2004]. Contudo, Chrysanthakopoulos & Singh [2005] e Russo [2006] observaram que, por mais promissor que seja a programação baseada em acordes, os problemas envolvidos em utilizar um compilador customizado fora de um ambiente estritamente experimental impede o acesso do "desenvolvedor comum" ao mundo dos acordes.

Por isso, Russo [2006] e Chrysanthakopoulos & Singh [2005] introduziram os acordes em C# por meio de uma biblioteca externa que poderia ser usada diretamente em um ambiente de desenvolvimento padrão. Liu [2008] tem progredido em iniciativa semelhante para C++.

Muito embora a implementação via biblioteca imponha diversas limitações tanto no que se refere à sintaxe quanto à possibilidade de otimização, essa abordagem disponibilizou a programação baseada em acordes a um público muito maior e viabilizou o seu uso mesmo em sistema legados.

Podemos destacar também o trabalho de Wood & Eisenbach [2004] no desenvolvimento do suporte a acordes para Java. Interessantemente esse trabalho foi baseado principalmente em Benton et al. [2002] e revisitou muitos dos conceitos vistos em Itzstein & Jasiunas [2003] sob uma nova ótica. Assim como Itzstein & Jasiunas [2003], essa solução também se baseou na criação de um compilador customizado para Java com suporte a acordes.

Contudo, nos últimos anos o desenvolvimento de acordes em Java foi significativamente deixado de lado. Até o momento, nem Itzstein [2005] nem Wood & Eisenbach [2004] disponibilizaram ao público os códigos fontes completos de seus trabalhos. Mesmo que esses códigos tivessem sido disponibilizados ambas as iniciativas são limitadas a ambientes experimentais, visto que a utilização de compiladores customizados impõe diversas limitações que não são aceitáveis para um universo considerável de potenciais desenvolvedores. Dentre elas podemos destacar:

- Suporte sub-ótimo à linguagem, em particular para funcionalidades menos utilizadas.
- Incompatibilidades com o ferramental existente (IDEs, debugadores, etc)
- Performance inferior tanto do código gerado como do próprio compilador
- Dificuldade e demora na incorporação correções, evoluções e inovações

Em princípio, mecanismos de concorrência podem ser implementados tanto como parte de linguagem quanto como bibliotecas externas. O benefício de se implementar tais mecanismos como parte da linguagem é a possibilidade dada ao compilador de realizar análises e otimizações mais aprofundadas [Benton et al., 2004], assim como disponibilizar uma sintaxe mais bem adaptada à tarefa.

Por outro lado, os acordes são uma criação relativamente recente. A evolução natural dessa construção, obtida por meio da experimentação prática e pesquisa teórica, pode levar a incrementos e alterações da mesma. Experimentação essa que só será possível se os acordes estiverem disponíveis a um público amplo e diversificado [Chrysanthakopoulos & Singh, 2005].

Ao contrário do que tem ocorrido em C#, a exemplo de Russo [2006], o desenvolvimento com acordes para Java está indisponível ao desenvolvedor médio e inacessível a sistemas legados. Não existe um meio incremental de incorporar acordes na base de software existente e mesmo a experimentação com acordes em Java é dificultada, haja vista o limitado público com a capacidade técnica para propor modificações e correções em compiladores customizados.

1.4 Objetivos

Considerando o exposto acima, esse trabalho tem os objetivos de:

- Estudar o impacto da implementação de acordes via biblioteca na arquitetura da linguagem Java
- Implementar acordes para Java por meio de uma biblioteca
- Implementar casos de uso práticos que utilizem essa biblioteca
- Avaliar a solução proposta considerando a base teórica de *join-calculus*
- Analisar comparativamente o uso de acordes em relação as primitivas de concorrência existentes em Java

Como pode ser visto no exemplo apresentado anteriormente, acordes, quando implementados diretamente na linguagem, são construções simples e elegantes que se encaixam bastante naturalmente na sintaxe da linguagem. Contudo, uma implementação de acordes via biblioteca traz diversos desafios que precisam ser superados.

Uma implementação de acordes, fundamentalmente, precisa possuir mecanismos que suportem a expressão de dois elementos chaves: métodos assíncronos e padrões *join* [Benton et al., 2007].

Os métodos assíncronos devem exprimir as noções de paralelismo e ausência de retorno de valor. Em Itzstein [2005] esses conceitos foram introduzidos por meio da palavra reservada `signal` que é usada como tipo de retorno para os métodos assíncronos (`async` em Benton et al. [2002]). Contudo, uma implementação de acordes via biblioteca não dispõe da possibilidade de introduzir novas palavras reservadas na linguagem.

Os padrões *join* devem exprimir as noções de sincronização e gestão de recursos que efetivamente formam as bases para os acordes. É importante observar que, na sintaxe padrão da linguagem Java, assim como diversas outras linguagens imperativas orientadas por objetos, uma declaração de método é feita com exatamente uma assinatura e exatamente um corpo. Essa não é uma sintaxe viável para acordes onde existe um conjunto de assinaturas de métodos que se referem a um único corpo.

Tanto Itzstein [2005] quanto Benton et al. [2004] expressam os padrões *join* via o operador "&" usado para ligar os diversos fragmentos de um acorde. Novamente, a introdução de um novo operador na linguagem não é uma opção viável no caso de uma implementação em biblioteca. Isso faz com que a sintaxe para declaração dos acordes seja uma construção não trivial nesse tipo de implementação, além de dificultar o *binding* dos parâmetros dos fragmentos do acorde que precisarão ser transportados para o corpo do mesmo.

Por fim, uma implementação de acordes via biblioteca precisa mapear as construções disponibilizadas nas primitivas de programação concorrentes disponíveis nativamente, tendo o cuidado de preservar as propriedades e características das abstrações de alto nível.

1.5 Organização do Texto

O Capítulo 2 contém a revisão dos fundamentos teóricos nos quais os acordes se baseiam. Começando com π-*Calculus*, passando pelo *Join-Calculus* e conclui detalhando o funcionamento básico dos Acordes propriamente ditos.

O Capítulo 3 apresenta a biblioteca JChords, desenvolvida nesse trabalho, que permite a programação baseada em acordes em Java. São apresentados os fundamentos da biblioteca e seu desenho. Além disso é feita uma análise comparativa entre JChords e Java padrão para casos de uso selecionados.

O Capítulo 4 detalha a arquitetura e implementação da biblioteca JChords. São analisados os diversos componentes da biblioteca e sua interação para prover acordes em Java em um ambiente de desenvolvimento padrão.

O Capítulo 5 faz uma análise comparativa da solução proposta com as principais soluções existentes.

Por fim o Capítulo 6 faz um apanhado geral do trabalho realizado e apresenta sugestões e desafios para trabalhos futuros.

Capítulo 2

Fundamentos Teóricos

Nesse capítulo são apresentados os fundamentos teóricos que suportam o desenvolvimento de sistemas baseados em acordes. É feita uma revisão das álgebras de processos, *π-calculus* e *join-calculus*, com o objetivo de mostrar como esses formalismos podem ser usados para prover abstrações elegantes para expressar paralelismo e concorrência. Por fim é formalizado o conceito de acorde e detalhadas as principais propriedades dessa construção no contexto de linguagens imperativas orientadas a objetos.

O *π-calculus* é uma álgebra de processos madura que resistiu à prova do tempo e mostra-se um formalismo robusto para expressar ou descrever o comportamento paralelo de processos concorrentes [Pierce, 1995]. Contudo, apesar de suas qualidades, *π-calculus* é limitado para expressar situações onde o envio e o recebimento de mensagens ocorre de maneira assíncronas e distribuída Fournet & Gonthier [2000].

Join-Calculus por sua vez é uma formalismo mais recente, desenvolvido como uma derivação assíncrona do *π-calculus*. Com isso buscou-se reusar as qualidades do *π-calculus* e ainda permitir expressar cenários que envolvem comunicações assíncronas entre processos distribuídos [Fournet & Gonthier, 2000].

Formalismos como esses permitem analisar e estudar as propriedades de modelos e mecanismos abstratos possibilitando criar um sólido entendimento de seu funcionamento. Porém, para serem efetivos fora do ambiente teórico é necessário transportar esses formalismos para técnicas, ferramentas e construções em ambientes de desenvolvimento.

Os acordes são construções adequadas à linguagens de programação imperativas orientadas a objetos, derivadas diretamente a partir do *join-calculus*. O acorde permite usar as estruturas do *join-calculus* como um mecanismo de orquestração de processos, permitindo expressar as regras de interação entre processos declarativamente. Com isso é possível abstrair o gerenciamento explícito das interações entre fluxos paralelos de execução em favor de conjuntos de regras impostas implicitamente.

2.1 π-*Calculus*

O π-*calculus* é uma álgebra de processos desenvolvido por Milner [1989] derivado de Milner [1980], Hoare [1985], entre outros. Trata-se de um modelo matemático para processos dinâmicos onde as interconexões entre processos mudam a medida que esses interagem entre si.

Em π-*calculus* cada expressão denota um processo, ou seja, uma atividade computacional executando em paralelo com outras atividades. O elemento fundamental de computação é a transmissão e recepção de mensagens através de canais de comunicação [Pierce, 1995].

Simplificadamente, π-*calculus* pode ser definido formalmente como:

Sintaxe

Nomes	$a, b, c, \ldots, x, y, z$	Canais	(2.1a)
Processos	$P, Q, R \equiv 0$	Processo inerte	(2.1b)
	$\parallel \bar{a}x.P$	Transmissão	(2.1c)
	$\parallel a(x).P$	Recepção	(2.1d)
	$\parallel P \mid Q$	Composição	(2.1e)
	$\parallel (\gamma x)P$	Restrição	(2.1f)
	$\parallel\ !P$	Replicação	(2.1g)

Congruência Estrutural

Composição (2.1e)	$P \mid 0 \equiv P$	(2.2a)
	$P \mid Q \equiv Q \mid P$	(2.2b)
	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(2.2c)
Restrição (2.1f)	$((\gamma x)P) \mid Q \equiv (\gamma x)(P \mid Q)$	(2.2d)
	$(\gamma x)(\gamma y)P \equiv (\gamma y)(\gamma x)P$	(2.2e)
	$(\gamma x)0 \equiv 0$	(2.2f)
Replicação (2.1g)	$!P \equiv P \mid !P$	(2.2g)

Semântica

$$\bar{a}x.P \mid a(y).Q \Rightarrow P \mid [x/y]Q \tag{2.3a}$$

π-*calculus* opera sobre nomes (2.1a) que podem ser canais de comunicação ou mensagens, sendo que canais podem ser usados como mensagens e vice-versa, ou seja, são apenas formas possíveis para a mesma entidade. Da mesma maneira, os nomes em si não são relevantes exceto como forma de identificar uma mesma entidade dentro da expressão, ou seja, a expressão $\bar{a}x$ é equivalente a $\bar{b}y$, e possuem o mesmo valor semântico.

A expressão 0 (2.1b) denota um processo inerte, podendo representar um processo inativo, ou ainda um processo sequencial sem nenhum comportamento concorrente observável [Pierce, 1995]. Sob essa perspectiva é interessante observar que π-*calculus* existe fundamentalmente no nível de orquestração de processos, sem considerar as atividades sequenciais internas de cada um dos processos [Parrow, 2001].

As expressões $\bar{a}x.P$ (2.1c) e $a(x).P$ (2.1d) denotam respectivamente processos que transmitem e recebem uma mensagem x pelo canal a, e depois se comportam como P. Tanto na transmissão quanto na recepção, o processo é bloqueado até que a comunicação tenha sido concluída, ou seja, a comunicação é síncrona [Pierce, 1995].

A expressão $P \mid Q$ (2.1e) denota a composição paralela de dois processos P e Q que são avaliados simultaneamente, possivelmente trocando mensagens entre si. A composição é o principal mecanismo que permite expressar comportamentos concorrentes em π-*calculus*. Observa-se pela equação (2.2a) que processos inertes podem ser desprezados sem comprometer o significado da expressão. Além disso a composição de processos é comutativa (2.2b) e associativa (2.2c).

A expressão de restrição $(\gamma x)P$ (2.1f) denota um processo que se comporta como P em um contexto em que x é um novo canal, diferente de qualquer outro canal presente em P. Essa expressão permite declarar explicitamente novos canais, independente da existência de outro canal que porventura venha a ter o mesmo nome.

A restrição permite extrusão de escopo (2.2d), ou seja, uma expressão de restrição pode ser movida para um escopo mais externo sem comprometer o significado da expressão, desde que observadas eventuais colisões de nomes. Restrições são comutativas (2.2e), além disso, uma restrição pode ser descartada se estiver sendo aplicada somente a um processo inerte (2.2f), tendo em vista que o novo canal declarado nunca seria usado.

Por fim, a expressão $!P$ (2.1g) denota um processo que se comporta como P e, em seguida, se comporta como $!P$ novamente, e assim por diante, indefinidamente. Como mostra (2.2g), uma replicação pode ser substituída por uma ocorrência do processo seguido pela própria replicação, efetivamente permitindo que uma expressão se estenda infinitamente quando necessário.

Como pode ser visto em (2.3a), semanticamente, toda a computação em *π-calculus* ocorre pela transmissão de mensagens de um processo para o outro. O exemplo abaixo ilustra um cenário produtor-consumidor:

$$\bar{a}x.x(w).0 \mid a(y).\bar{y}z.0 \tag{2.4a}$$

$$x(w).0 \mid \bar{x}z.0 \tag{2.4b}$$

$$0 \mid 0 \tag{2.4c}$$

Nesse exemplo um processo transmite x pelo canal a, aguarda uma mensagem w pelo próprio canal x e então fica inerte. Enquanto isso, em paralelo, outro processo aguarda uma mensagem y pelo canal a, transmite z pelo canal y recebido e então fica inerte [Parrow, 2001].

No primeiro passo (2.4a) ocorre a transmissão de x de um processo para o outro, conforme a equação (2.3a). Com isso, no processo receptor, o canal y é substituído por x. Tanto a expressão de transmissão quanto a recepção são consumidas.

No segundo passo (2.4b) o canal x (originalmente chamado y) é usado para transmitir z para o outro processo que aguarda a recepção de w nesse canal. Assim w no processo receptor será substituído por z e tanto a expressão de transmissão quanto a de recepção são consumidas.

O que nos leva ao terceiro e último passo (2.4c) onde restam somente dois processos inertes, não restando mais nenhuma computação possível.

π-calculus, assim como outras álgebras de processo, busca prover ao campo da programação paralela e concorrente o mesmo tipo de rigor e formalismo que *λ-calculus* provê à programação sequencial. Dispondo de um conjunto mínimo de construções, *π-calculus* permite o estudo formal das propriedades fundamentais de sistemas concorrentes.

Vale destacar também que *π-calculus* tem o mesmo poder computacional de *λ-calculus*, ou seja, *π-calculus* é Turing completa e representa um modelo universal de computação [Milner, 1992]. Além disso, há um extenso corpo de trabalho relacionado a *π-calculus* sendo um modelo bastante utilizado e estudado.

Contudo, *π-calculus* se baseia em canais de comunicação síncronos, ou seja as operações de envio e recebimento de uma mensagem encapsulam a transmissão, recepção,

roteamento e sincronização dessa mensagem de maneira atômica [Fournet & Gonthier, 1996].

Em um ambiente onde os elementos computacionais podem estar espacialmente distantes, onde problemas de falhas e latências precisam ser considerados, garantir uma interação atômica entre transmissor e receptor pode não ser uma opção viável. Assim encapsular essa interação como uma operação atômica passa a ser indesejável. Muito embora isso possa ser tratado em π-*calculus* considerando o meio de transmissão como um processo, essa técnica pode aumentar a complexidade das equações.

2.2 *Join-Calculus*

Join-calculus foi desenvolvido como uma derivação assíncrona do π-*calculus* onde os conceitos de transmissão e sincronização estão desacoplados de modo que toda a sincronização possa ser feita localmente. Ou seja, trata-se de uma tentativa de prover uma álgebra de processos para comunicação assíncrona e distribuída, baseada em um modelo formal bem definido e com propriedades conhecidas [Fournet & Gonthier, 2000].

O *join-calculus* é, em essência, a implementação algébrica de uma Máquina Abstrata Química Reflexiva ou RCHAM (*Reflexive CHemical Abstract Machine*) [Wood & Eisenbach, 2004]. A RCHAM é um modelo de máquina abstrata proposto por Fournet & Gonthier [1996] que introduz os conceitos de localidade e reflexão na CHAM (CHemical Abstract Machine) proposta por Berry & Boudol [1992].

Esse modelo utiliza uma metáfora química para modelar processos concorrentes. Nessa metáfora, o estado do sistema é representado por uma solução química composta por um multiconjunto de átomos em suspensão. Essa solução é governada por um conjunto de regras de reação que definem como os átomos irão interagir dentro da solução [Berry & Boudol, 1992].

O conceito de átomo nesse modelo difere em alguns aspectos do seu conceito tradicional na química. A palavra átomo é usada aqui no sentido literal de uma partícula indivisível capaz de participar em reações químicas. Outras propriedades químicas como estrutura atômica, valência ou conservação de massa não são relevantes para o modelo.

Considere como exemplo um dado sistema composto pelos átomos $\{A, B, C\}$ e

regido pelas regras:

$$R_1 \equiv B \,\&\, C \triangleright D, E \tag{2.5a}$$
$$R_2 \equiv X \,\&\, E \triangleright F \tag{2.5b}$$
$$R_3 \equiv Y \,\&\, A \triangleright G \tag{2.5c}$$
$$R_4 \equiv Y \,\&\, D \triangleright H \tag{2.5d}$$

Onde, informalmente, o operador & pode ser interpretado como "reage com" e o operador $\triangleright$ pode ser interpretado como "produz". Dado nosso sistema exemplo, vemos que a solução possui todos os átomos necessários para reação R_1 (2.5a). Desse modo, ao ser desencadeada, a reação R_1 (2.5a) consome os átomos B e C, substituindo-os pelos átomos D e E. Essa reação é demonstrada na Figura 2.1.

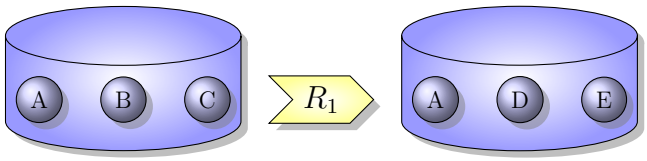

Figura 2.1. Uma reação na RCHAM

Nesse ponto, podemos observar que não há mais nenhuma regra de reação que possa ser aplicada. Dessa forma consideramos a solução inerte, e ela permanecerá assim até que novos átomos sejam adicionados à mesma.

Introduz-se então na solução o átomo X. A introdução de X irá desencadear a reação R_2 que irá consumir os átomos E e X é ira produzir o átomo F. Essa reação é demonstrada na Figura 2.2.

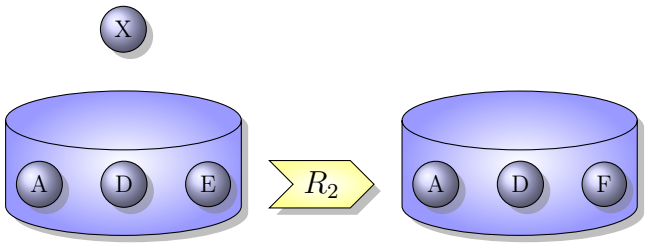

Figura 2.2. Adição de um átomo na RCHAM

Novamente, a solução está em um estado inerte e permanecerá assim até que novos átomos sejam introduzidos.

Por fim, introduz-se o átomo Y. Como pode ser observado, a introdução do átomo Y tem o potencial de desencadear duas reações distintas, R_3 e R_4. Conforme Fournet & Gonthier [2000], por definição, um átomo só pode ser consumido por uma única

reação. A estratégia de escolha de qual reação deverá ocorrer não é parte do modelo e é deixado como uma decisão de implementação. Dessa forma, embora não seja possível prever no nosso exemplo qual reação irá ocorrer, podemos afirmar que apenas uma das duas reações possíveis irá ocorrer. Essa reação é demonstrada na Figura 2.3

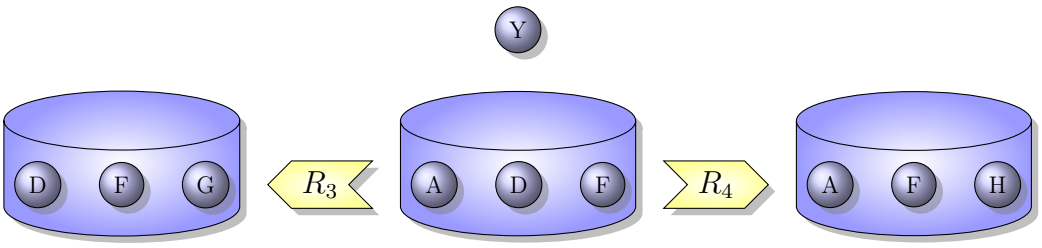

Figura 2.3. Reação não-determinística da RCHAM

Novamente, deixando a solução em um estado inerte.

Uma expressão em *join-calculus* é composta por processos, padrões *join* e definições. Como o π-*calculus* todos os valores são nomes. A notação $\tilde{x}$ representa uma tupla $x_1, x_2, x_3, \ldots, x_n$ e o operador *join* é representado pelo símbolo &. Dessa forma Odersky [2000] define formalmente *Join-Calculus* como:

Sintaxe

Nomes	$a, b, c, \ldots, x, y, z$	Canais	(2.6a)
Processos	$P, Q, R \equiv x(\tilde{v})$	Transmissão	(2.6b)
	$\parallel \text{def } D; P$	Definição	(2.6c)
	$\parallel P \,\&\, Q$	Composição	(2.6d)
Padrões *join*	$J, K \equiv x(\tilde{v})$	Mensagem	(2.6e)
	$\parallel J \,\&\, K$	*Join*	(2.6f)
Definições	$D, E, F \equiv J = P$	Reação	(2.6g)
	$\parallel D, E$	Definição composta	(2.6h)

Congruência Estrutural

Definição (2.6c)	$\text{def } D; \text{def } E; P \equiv \text{ def } D, E; P$	(2.7a)
	$(\text{def } D; P) \,\&\, Q \equiv \text{ def } D; P \,\&\, Q$	(2.7b)
Composição (2.6d)	$P \,\&\, Q \equiv Q \,\&\, P$	(2.7c)
	$P \,\&\, (Q \,\&\, R) \equiv (P \,\&\, Q) \,\&\, R$	(2.7d)
Definição composta (2.6h)	$D, E \equiv E, D$	(2.7e)
	$D, (E, F) \equiv (D, E), F$	(2.7f)

Semântica

$$\text{def } J = P, D; J \,\&\, Q \Rightarrow \text{ def } J = P, D; P \,\&\, Q \tag{2.8a}$$

Um processo pode ser a emissão de uma mensagem (2.6b), uma definição de um novo padrão *join* (2.6c) ou uma composição de processos (2.6d), sendo que a composição de processos é comutativa (2.7c) e associativa (2.7d).

A emissão de uma mensagem (2.6b) disponibiliza a mesma para reagir com outras mensagens de acordo com as definições existentes na expressão. Uma mensagem permanece na expressão até que forme um padrão *join*.

Uma vez que os processos são compostos (2.6d) usando o operador *join*, temos como efeito que o envio de uma mensagem não requer sua recepção imediata. É possível continuar a avaliar a expressão mesmo que uma mensagem permaneça nesta expressão indefinidamente. Dessa forma, a emissão de uma mensagem constitui uma operação assíncrona dentro do *join-calculus*.

Em um processo, duas ou mais definições podem ser agregadas, formando uma única definição composta (2.7a). Além disso as definições permitem extrusão de escopo (2.7b) de modo que o escopo da definição pode ser estendido dinamicamente para incorporar outros elementos. Em ambos os casos devem ser observadas eventuais colisões de nomes.

Um padrão *join* é formado por uma mensagem (2.6e) ou um conjunto de mensagens ligadas pelo operador *join* (2.6f). Esse padrão descreve quais mensagens um processo deve emitir para desencadear uma reação (2.6g).

Por sua vez, uma definição, denota uma regra de reação (2.6g) ou uma composição de regras de reação (2.6h), sendo que essas definições compostas são comutativas (2.7e)

e associativas (2.7f).

Uma regra de reação (2.6g) descreve uma possível redução da expressão (2.8a). Uma regra de reação é formada por um padrão *join* e um processo. Sempre que as mensagens presentes em uma expressão formem o padrão *join* de uma regra de reação, então essas mensagens podem ser substituídas pelo processo dessa regra. A avaliação de uma expressão em *join-calculus* se dá pela aplicação das regras de reação sobre os possíveis padrões *join* presentes na mesma, sendo que as expressões não reagem até que todas as mensagens de algum padrão *join* tenham sido emitidas [Itzstein, 2005].

Dada uma expressão qualquer, só existem duas reduções possíveis [Fournet & Gonthier, 2000]:

- Enviar uma mensagem por um canal
- Aplicar uma reação cujo padrão *join* esteja presente na expressão

Considere o exemplo de uma fila de impressão adaptado de Fournet & Gonthier [1996]:

Seja:

$$D \equiv ready(p)\ \&\ job(j) = print(p, j) \tag{2.9a}$$

$$E \equiv print(p, j) = ready(p) \tag{2.9b}$$

São definidas acima duas regras de reação denominadas D (2.9a) e E (2.9b).

A primeira regra é composta pelo padrão *join* $ready(p)\ \&\ job(j)$ e o processo $print(p, j)$ e denota que, sempre que as mensagens $ready(p)$ e $job(j)$ estiverem presentes na expressão, essas podem ser substituídas pela mensagem $print(p, j)$. A mensagem $ready(p)$ indica que a impressora p está disponível para impressão, a mensagem $job(j)$ indica que existe um trabalho de impressão j a ser realizado e a mensagem $print(p, j)$ indica que o trabalho de impressão j será impresso pela impressora p. Dessa forma a regra D diz que sempre que houver uma impressora disponível e um trabalho de impressão a ser realizado, então esse trabalho deve ser enviado para ser impresso nessa impressora.

A segunda regra, composta pelo padrão *join* $print(p, j)$ e pelo processo $ready(p)$, diz que sempre que uma impressora p tiver realizado um trabalho de impressão j, o que é indicado pela existência de uma mensagem $print(p, j)$ na expressão, então essa

impressora deverá ser disponibilizada para novos trabalhos de impressão, por meio da mensagem $ready(p)$.

Assim considere um sistema de fila de impressão que possui as regras D e E, uma impressora p_1 e no qual existam 2 trabalhos de impressão a ser realizados, j_1 e j_2:

$$
\begin{aligned}
spooler &\equiv \underline{\text{def } D; \text{def } E;} ready(p_1) \,\&\, job(j_1) \,\&\, job(j_2) && [\text{ por (2.7a) }] && (2.10a)\\
&\equiv \text{def } D, E; \underline{ready(p_1) \,\&\, job(j_1)} \,\&\, job(j_2) && [\text{ por (2.9a) }] && (2.10b)\\
&\equiv \text{def } D, E; \underline{print(p_1, j_1)} \,\&\, job(j_2) && [\text{ por (2.9b) }] && (2.10c)\\
&\equiv \text{def } D, E; \underline{ready(p_1) \,\&\, job(j_2)} && [\text{ por (2.9a) }] && (2.10d)\\
&\equiv \text{def } D, E; \underline{print(p_1, j_2)} && [\text{ por (2.9b) }] && (2.10e)\\
&\equiv \text{def } D, E; ready(p_1) && && (2.10f)
\end{aligned}
$$

No primeiro passo as regras D e E são compostas em uma única regra composta D, E apenas como uma forma de higienizar a expressão e deixá-la mais simples.

No próximo passo a regra D é aplicada sobre as mensagens $ready(p_1)$ e $job(j_1)$ substituindo-as pela mensagem $print(p_1, j_1)$, o que efetivamente envia o trabalho de impressão j_1 para ser impresso na impressora p_1.

O próximo passo aplica a regra E à mensagem $print(p_1, j_1)$ emitida no passo anterior e a substitui pela mensagem $ready(p_1)$ o que re-disponibiliza a impressora p_1 para novos trabalhos de impressão.

O próximo passo aplica novamente a regra D, dessa vez às mensagens $ready(p_1)$ e $job(j_2)$ que são substituídas na expressão pela mensagem $print(p_1, j_2)$ que envia o trabalho j_2 para impressão em p_1.

O próximo e último passo aplica novamente a regra E e re-disponibiliza p_1 para novos trabalhos de impressão pela emissão da mensagem $ready(p_1)$.

Vale relembrar que o *join-calculus*, assim como o π-*calculus*, opera no nível de orquestração de processos. Em uma implementação real a mensagem $print(p, j)$ provavelmente desencadearia um processo sequencial que faria acesso ao *hardware* da impressora e executaria a impressão propriamente dita.

É importante observar também que o mesmo conjunto de regras de reação é aplicável para qualquer número de impressoras e trabalhos de impressão. Os recursos compartilhados, no caso as impressoras e os trabalhos de impressão, são gerenciados pela própria estrutura de mensagens do *join-calculus* que garante a alocação apropriada dos mesmos de acordo com as regras de reação. Isso pode ser melhor observado

no exemplo a seguir onde são utilizadas duas impressoras simultaneamente:

$$\equiv \text{def } D, E; ready(p_1) \,\&\, \underline{ready(p_2) \,\&\, job(j_1)} \,\&\, job(j_2) \quad [\text{ por (2.7c) }] \quad (2.11a)$$
$$\&\, jobj_3$$
$$\equiv \text{def } D, E; \underline{ready(p_1) \,\&\, job(j_1)} \,\&\, \underline{ready(p_2) \,\&\, job(j_2)} \quad [\text{ por } 2 \times (2.9a)] \quad (2.11b)$$
$$\&\, jobj_3$$
$$\equiv \text{def } D, E; \underline{print(p_1, j_1)} \,\&\, \underline{print(p_2, j_2)} \,\&\, job(j_3) \quad [\text{ por } 2 \times (2.9b)] \quad (2.11c)$$
$$\equiv \text{def } D, E; ready(p_1) \,\&\, \underline{ready(p_2) \,\&\, job(j_3)} \quad [\text{ por (2.9a) }] \quad (2.11d)$$
$$\equiv \text{def } D, E; ready(p_1) \,\&\, \underline{print(p_2, j_3)} \quad [\text{ por (2.9b) }] \quad (2.11e)$$
$$\equiv \text{def } D, E; ready(p_1) \,\&\, ready(p_2) \quad (2.11f)$$

Nota-se que tanto em (2.11b) quanto em (2.11c) existe a aplicação simultânea das regras de reação. Apesar disso, em ambos os casos, a alocação dos recursos compartilhados, é feita de forma atômica. As mensagens que compõem um padrão *join* e os recursos que essas mensagens encapsulam são disponibilizados exclusivamente para uma única regra de reação.

Apesar de suas diferenças, o *join-calculus* é uma variação assíncrona do π-*calculus* que possui a mesma capacidade de expressão [Fournet & Gonthier, 2000]. Formalmente o *join-calculus* conserva todas a propriedades do π-*calculus*, e acrescenta as seguintes restrições:

- As construções para restrição de escopo, recepção e replicação são combinadas em uma única construção, a definição de regras;
- A comunicação só ocorre por meio de nomes definidos;
- para cada nome definido existe exatamente uma replicação.

Join-Calculus se apresenta como um formalismo expressivo e poderoso para expressar computações paralelas e concorrentes. Sua estrutura dispõe fundamentalmente de processos que emitem mensagens. Essas mensagens são interpretadas e avaliadas por regras de reação. Cada regra de reação, por sua vez, consome as mensagens que compõem o seu padrão *join*, evitando assim que mais de uma regra seja aplicada simultaneamente à mesma mensagem e garantindo que não haja o compartilhamento indevido de recursos.

A analogia entre o *join-calculus*, composto por processos emitindo mensagens, e o paradigma orientado a objetos, onde toda a computação é realizada por meio de

objetos emitindo mensagens, é bastante interessante. Adiciona-se a isso a pequena diferença conceitual que separa a maneira como, no paradigma orientado a objetos, a emissão de uma mensagem desencadeia a execução de um método e, em *join-calculus*, a emissão de um conjunto de mensagens desencadeia uma regra de reação. Tudo isso torna o *join-calculus* um bom candidato para prover as bases para abstrações de alto nível para programação concorrente em linguagens orientadas a objeto de propósito geral [Itzstein, 2005].

2.3 Acordes

O acorde é uma construção de alto nível de abstração que permite modelar soluções para problemas de concorrência e sincronização em linguagens imperativas orientadas a objetos de uma maneira elegante e efetiva.

Acordes integram a estrutura conceitual do *join-calculus* ao paradigma imperativo orientado a objeto, permitindo expressar declarativamente por meio de padrões *join* os relacionamentos de concorrência e as interações entre fluxos de execução paralelos.

Dessa forma as interações entre fluxos de execução paralelos são coordenados de maneira implícita, usando as regras de reação do *join-calculus* como mecanismo de orquestração de processos. Ou seja, sistemas baseados em acordes não requerem o gerenciamento explícito de fluxos de execução paralelos e de suas interações, sendo que interações paralelas ocorrem sempre de acordo com as regras previamente declaradas.

Um sistema baseado em acordes consiste em um conjunto de classes dentre as quais uma ou mais classes possuem pelo menos um acorde. Um acorde é composto por uma assinatura e um corpo. O corpo de um acorde é um bloco de código, composto por um conjunto de ações ou comandos. A assinatura de um acorde por sua vez consiste em um conjunto de fragmentos de acordes ligados pelo operador *join*. Um fragmento de acorde pode ser descrito como uma assinatura de método ou mensagem, sendo que um dado fragmento pode ser síncrono ou assíncrono [Petrounias et al., 2008].

Será utilizada a seguir uma sintaxe simplificada para acordes, derivada de Java, baseada no denominador comum de Benton et al. [2002] e Itzstein [2005]. Os modificadores de acesso como `public` e `private` são omitidos. Fragmentos assíncronos não possuem o tipo de retorno. O operador *join* será representado pelo símbolo "&". Por convenção, os fragmentos assíncronos, se existirem, serão sempre os primeiros do acorde. Todas as demais construções mantêm seu significado usual.

Utilizando essa notação, uma classe que utiliza acordes pode ser descrita como mostra a Listagem 2.1

```
1 class ChordedClass {
```

```
2   int sync(int p1) & async1(int p2, int p3) & async2() {
3     return p1 + p2 + p3;
4   }
5 }
```

Listagem 2.1. Estrutura de um acorde

No exemplo acima temos a classe com acordes `ChordedClass` que possui um único acorde cuja assinatura é composta pelo fragmento síncrono `sync(int p1)` e os fragmentos assíncronos `async1(int p2, int p3)` e `async2()` e cujo corpo retorna a soma dos parâmetros `p1`,`p2` e `p3`.

Uma instância de uma classe com acordes equivale a uma expressão em *join-calculus*. De maneira análoga um fragmento de acorde equivale a uma mensagem (2.6e) e a invocação desse fragmento equivale ao envio dessa mensagem (2.6b). Cada acorde equivale a uma definição (2.6c) de uma regra de reação (2.6g), sendo que a assinatura do acorde equivale ao padrão *join* e o corpo equivale ao processo que a reação produz.

Um acorde possui no máximo um fragmento síncrono e quantos fragmentos assíncronos quanto necessário. O tipo de retorno do fragmento síncrono determina o tipo de dado retornado pela execução do acorde. Fazem parte do escopo do corpo do acorde todos os parâmetros passados a cada um dos fragmentos do acorde [Wood & Eisenbach, 2004].

Sempre que um fragmento de um acorde é invocado em uma dada instância da classe que define esse acorde, considera-se que passa a existir uma invocação pendente para aquele fragmento na referida instância. A execução de um acorde ocorre quando há pelo menos uma invocação pendente para cada um dos fragmentos que compõem o acorde. Cada execução do acorde consome uma invocação pendente de cada um dos fragmentos daquele acorde [Petrounias et al., 2008].

A invocação de um fragmento assíncrono não retém o fluxo de execução que o invoca. Por sua vez a invocação de um fragmento síncrono retém o fluxo de execução até que uma execução de acorde consuma a invocação pendente desse fragmento síncrono [Wood & Eisenbach, 2004].

Seja `x` uma instância da classe `ChordedClass` apresentada na Listagem 2.1, considere a sequência de comandos apresentados na Tabela 2.1.

Nesse exemplo temos um total de sete invocações, sendo que a quarta e sétima invocação causam a execução do acorde. Sempre que o acorde é executado é consumida uma invocação pendente de cada fragmento. No fim uma invocação é deixada pendente. Vale ressaltar que, assim como em *join-calculus*, não há uma regra definida para qual invocação pendente a execução do acorde irá consumir, ou seja, a invocação pendente no fim pode ser qualquer uma das três invocações a `x.async2()`.

	Invocações Pendentes			Executa
Comando	sync	async1	async2	Acorde
`x.async2()`	0	0	1	-
`x.async1(1,2)`	0	1	1	-
`x.async1(3,4)`	0	2	1	-
`x.sync(5)`	0	1	0	sim
`x.async2()`	0	1	1	-
`x.async2()`	0	1	2	-
`x.sync(6)`	0	0	1	sim

Tabela 2.1. Invocações

Partindo dessas definições podemos revisitar o exemplo da fila de impressão visto anteriormente. Temos um conjunto de impressoras representadas pela classe `Printer`. Temos um conjunto de trabalhos de impressão representados pela classe `Job`. Cada trabalho de impressão deve ser impresso em uma das impressoras uma única vez e uma impressora é capaz de realizar um único trabalho de impressão por vez. Será criada uma fila de impressão, representada pela classe `Spooler` descrita na Listagem 2.2, responsável por gerenciar as interações entre impressoras e trabalhos de impressão.

```
1  class Spooler {
2    ready(Printer p) & job(Job j) {
3      print(p, j);
4    }
5
6    print(Printer p, Job j) {
7      ready{p};
8    }
9  }
10
11 Spooler spooler = new Spooler();
12 Printer p1 = new Printer();
13 Job j1 = new Job();
14 Job j2 = new Job();
15
16 spooler.ready(p1);
17 spooler.job(j1);
18 spooler.job(j2);
```

Listagem 2.2. Fila de impressão

A classe `Spooler` é composta por dois acordes. O primeiro acorde, cujo padrão *join*

é composto pelos fragmentos assíncronos `ready(Printer)` e `job(Job)`, é responsável por enviar trabalhos de impressão para as impressoras disponíveis. O segundo, contendo somente o fragmento assíncrono `print(Printer, Job)`, re-disponibiliza a impressora para novos trabalhos.

Após a declaração da classe com acordes temos a instanciação de quatro objetos, um objeto da classe `Spooler` chamado `spooler` que representa a fila de impressão. Um objeto da classe `Printer` chamado `p1` que representa uma impressora. E por fim dois objetos da classe `Job` chamados `j1` e `j2` que representam dois trabalhos de impressão.

Esse sistema realiza três invocações de fragmentos de acordes, ou seja, ele envia três mensagens para a fila de impressão. A primeira é uma mensagem assíncrona `ready(p1)`, que disponibiliza a impressora `p1` para a fila de impressão. Essa invocação não causa a execução do acorde e, sendo assíncrona, não bloqueia o fluxo de execução do sistema.

A segunda mensagem, `job(j1)`, requer a impressão do trabalho `j1` e a terceira mensagem, `job(j2)`, requer a impressão do trabalho `j2`. Sendo ambas assíncronas, nenhuma dessas mensagens retém o fluxo de execução do sistema.

Paralelamente `ready(p1)` e `job(j1)` causam a execução desse acorde definido na fila de impressão que, por sua vez irá invocar `print(p1, j1)`. Ou seja, `j1` será enviado para impressão em `p1`. Após a impressão, a mensagem `ready(p1)` será enviada novamente à própria fila de impressão, pelo acorde `print(Printer, Job)`, disponibilizando a impressora `p1` para novos trabalhos.

Isso causa nova execução do acorde devido ao *join* com a mensagem `job(j2)` que havia sido invocada no fim do fluxo principal do sistema. Assim `j2` também será enviado para impressão e em seguida, novamente, `ready(p1)` disponibilizará `p1` para futuras impressões.

O código da fila de impressão apresentado não possui nenhuma sincronização ou *pooling* explícitos. É escalonável para qualquer número de impressoras e trabalhos de impressão. Nenhum trabalho de impressão é enviado a uma impressora ocupada, nenhum trabalho é impresso mais de uma vez e todos os trabalhos serão impressos se houver impressoras disponíveis.

Os acordes permitem expressar relações concorrentes complexas de maneira simples, direta e declarativa. Isso facilita a criação de sistemas altamente paralelos de maneira clara e objetiva sem a necessidade de gerenciar explicitamente todas as interações possíveis entre os processos.

A programação baseada em acordes apresenta diversas semelhanças com várias outras técnicas e metodologias empregadas no desenvolvimento de sistemas paralelos e concorrentes.

O acorde se assemelha, sob certos aspectos, a um conjunto de semáforos interliga-

dos, ou ainda, a uma forma restrita de comandos guardados [Dijkstra, 1975]. Sendo que o acorde provê, adicionalmente, um mecanismo de gestão implícita de recursos compartilhados.

O modelo estrutural de acordes apresenta também diversas semelhanças com modelos baseados em espaços de tuplas como Linda [Valente, 2002]. Sendo que acordes se destina principalmente a coordenação de processos leves, ao passo que Linda se destina principalmente a coordenação de processos pesados.

A programação baseada em acordes é um novo paradigma para o desenvolvimento de sistemas paralelos e concorrentes. É possível demonstrar que muitas das construções de sincronização normalmente usadas na programação baseada em *threads* pode ser codificada por meio de acordes.

A exclusão mútua ou *mutex* é uma construção ou algoritmo que impede que dois processos concorrentes executem uma dada porção de código. O *mutex* deve garantir que em um dado momento, somente um processo tenha acesso à região crítica. *Mutexes* podem ser implementados quase que diretamente usando acordes, como pode ser visto na Listagem 2.3.

```
1  class Mutex {
2    Mutex() {
3      unlock();
4    }
5
6    void lock() & unlock() {
7      criticalRegion();
8      unlock();
9    }
10   ...
11 }
```

Listagem 2.3. Mutex

Como pode ser observado na Listagem 2.3, a classe `Mutex`, ao ser instanciada, invoca o fragmento assíncrono `unlock()`. A existência de uma chamada pendente a esse fragmento indica que nenhum processo está executando a região crítica. O fragmento síncrono `lock()` é invocado sempre que um determinado processo precisa executar a região crítica.

Se existe uma chamada pendente à `unlock()` e um processo solicita a execução da região crítica por meio do fragmento `lock()`, então o padrão *join* `lock()& unlock()` irá consumir os dois fragmentos e causar a execução do acorde que irá executar a região crítica e, em seguida, invocar novamente o fragmento `unlock()`. Por outro lado, a

inexistência de uma chamada pendente à `unlock()` fará com que qualquer processo que invoque `lock()` seja bloqueado até que uma chamada a `unlock()` seja feita, o que, como já foi citado, só ocorre no momento em que um processo deixa a região crítica.

Semáforos são, em essência, uma generalização dos *mutexes*. Enquanto um *mutex* permite que um, e somente um, processo entre na região crítica, semáforos devem permitir que um certo número de processos acessem a região crítica simultaneamente.

A implementação de semáforos por meio de acordes, mostrada na Listagem 2.4, é bastante semelhante à implementação de *mutexes* exceto pelo fato que a instanciação de um semáforo irá invocar o fragmento `unlock()` tantas vezes quanto o número de processos que podem acessar simultaneamente a região crítica.

```
1  class Semaphore {
2    Semaphore(int n) {
3      for(int index = 0; index < n; ++index) {
4        unlock();
5      }
6    }
7
8    void lock() & unlock() {
9      criticalRegion();
10     unlock();
11   }
12 }
```

Listagem 2.4. Semáforo

Ao ser instanciado o semáforo invoca o fragmento assíncrono `unlock()`, `n` vezes. O número de chamadas pendentes a esse fragmento indica quantos processos ainda podem entrar na região crítica. Cada invocação ao fragmento síncrono `lock()` permite ou bloqueia, de acordo com o padrão *join* `lock()& unlock()`, a um determinado processo o acesso à região crítica.

Hoare [1974] propôs monitores como forma de sincronização de processos que consiste em estabelecer um *token* para cada recurso compartilhado sendo que somente o processo que detém o *token* para um determinado recurso poderá acessá-lo. Os monitores são hoje um dos principais mecanismos de sincronização adotados por linguagens de programação orientadas a objetos.

Em acordes o *token* pode ser representado por um fragmento assíncrono que é necessário para executar os acordes correspondentes ao recurso compartilhado.

```
1  class Monitor {
2    Monitor() {
```

```
3     unlock();
4   }
5   method1() & unlock() {
6     runMethod1();
7     unlock();
8   }
9   method2() & unlock() {
10    runMethod2();
11    unlock();
12  }
13  ...
14  methodN() & unlock() {
15    runMethodN();
16    unlock();
17  }
18 }
```

Listagem 2.5. Monitor

No exemplo mostrado na Listagem 2.5 o fragmento `unlock()` representa o *token* para as instâncias da classe `Monitor`. Ao ser instanciada, a classe `Monitor` invoca o fragmento `unlock()`, disponibilizando o *token* para qualquer processo que o requeira. Quando um processo invoca um dos métodos de `Monitor` o acorde relativo a esse método é executado, consumindo o *token*. Por consequência, qualquer outra invocação a um dos métodos de `Monitor` bloqueia o fluxo de execução até que uma nova chamada a `unlock()` seja realizada, o que deve ocorrer no fim da execução de cada um dos acordes.

Temos também a barreira, um mecanismo de sincronização que cria um "ponto de encontro" que sincroniza a execução de diversos processos. Cada processo que atinge a barreira deve ser bloqueado até que todos os processos, ou um subconjunto especificado, tenham atingido a barreira. Nesse momento todos os processos são desbloqueados simultaneamente.

Em acordes, uma barreira pode ser implementada como mostra a Listagem 2.6.

```
1 class Barrier {
2   Barrier(int n) {
3     for(int index = 0; index < n; ++index)
4     {
5       process(index);
6     }
7     enter(n);
8     between();
```

```
9       leave(n);
10    }
11
12    async process() {
13      before();
14      ready();
15      after();
16    }
17
18    void enter(int n) {
19      expectEnter(n);
20      waitEnter();
21    }
22
23    void leave(int n) {
24      expectLeave(n);
25      waitLeave();
26    }
27
28    void ready() & expectEnter(int n) {
29      if (--n > 0) {
30        expectEnter(n);
31      } else {
32        allReady();
33      }
34      gone();
35    }
36
37    void gone() & expectLeave(int n){
38      if (--n > 0) {
39        expectLeave(n);
40      } else {
41        allGone();
42      }
43    }
44
45    void waitEnter() & allReady() {}
46    void waitLeave() & allGone() {}
47
48  }
```

Listagem 2.6. Barreira

No exemplo da Listagem 2.6, cada processo é representado por uma chamada ao método assíncrono `process(int)`. Cada processo invoca o método `before()`, que é executado antes de atingir a barreira, seguido pelo fragmento síncrono `ready()`, que representa a barreira propriamente dita, seguido pelo método `after()`, que é executado após a barreira.

Ao ser instanciada a classe `Barrier` inicia cada um dos processos e, em seguida invoca o método `enter(int)`, especificando o número de processos que a barreira deve reter. O método `enter(int)` invoca o fragmento síncrono `waitEnter()`, que bloqueia até que todos os processos tenham entrado na barreira.

Na sequência o método `between()` é executado, o que corresponde ao momento em que todos os processos estão bloqueados na barreira e antes de qualquer um deles ser liberado para continuar a executar.

Por fim a chamada ao método `leave(int)` libera os processos para que continuem sua execução. Análogo ao método `enter(int)`, `leave(int)` invoca o fragmento síncrono `waitEnter()` que bloqueia até que todos os processos tenham saído da barreira.

O acorde `ready()& expectEnter(int)` sincroniza a entrada dos processos na barreira, por meio do fragmento síncrono `ready()` e pelo número de processos especificado pelo fragmento assíncrono `expectEnter(int)`. Quando todos os processos atingem a barreira esse acorde invoca o fragmento assíncrono `allReady()` que por sua vez permite a execução do acorde `waitEnter()& allReady()`, o que irá desbloquear a invocação ao fragmento síncrono `waitEnter()` feita pelo método `enter(int)` na instanciação da classe `Barrier`.

De maneira análoga o acorde `void gone()& expectLeave(int)` sincroniza a saída dos processos da barreira por meio do fragmento síncrono `gone()` e pelo número de processos especificado em `expectLeave(int)`. Quando todos os processos saírem da barreira esse acorde invoca o fragmento assíncrono `allGone()` que permite a execução do acorde void `waitLeave()& allGone()` que desbloqueia o fragmento síncrono `waitLeave()` invocado pelo método `leave(int)` na instanciação da classe `Barrier`.

É importante destacar que, muito embora os mecanismos de sincronização tradicionais possam ser emulados por meio de acordes, este não é o uso mais adequado dessa técnica, uma vez que os acordes oferecem um repertório diversificado e flexível de padrões de sincronização que podem ser aplicados de maneira mais específica a cada situação.

Em resumo, álgebras de processo como *join-calculus* são amplamente usadas para analisar, especificar e verificar sistemas paralelos e concorrentes [Pierce, 1995] Fournet

& Gonthier [1996]. O acorde, sendo derivado diretamente do *join-calculus*, se beneficia amplamente dessa solidez conceitual e integra-se bem no paradigma imperativo orientado a objetos, oferecendo uma forma elegante e eficaz de solucionar uma gama de problemas práticos de concorrência e paralelismo.

2.4 Conclusões

Formalismos como π-*calculus* e *join-calculus* fornecem abstrações elegantes para expressar paralelismo e concorrência. O π-*calculus* é uma álgebra de processos amplamente utilizada para descrever o comportamento paralelo de processos concorrentes [Pierce, 1995]. *Join-Calculus* por sua vez é uma derivação assíncrona do π-*calculus* que equivale ao π-*calculus* em termos de poder computacional e simplifica a expressão de cenários que envolvem comunicações assíncronas entre processos distribuídos [Fournet & Gonthier, 1996].

O acorde é uma forma de implementar as construções do *join-calculus* em linguagens de programação imperativas orientadas a objetos, que busca transpor da esfera teórica para a prática do desenvolvimento de *software* a elegância conceitual desta algebra de processos.

Os acordes fornecem um mecanismo de orquestração de processos que permite expressar as regras de interação entre processos declarativamente. Os acordes especificam padrões *join* que descrevem padrões de mensagens necessárias à execução de determinadas seções de código, encapsulando assim o protocolo de execução dessas seções, assim como a gestão de recursos compartilhados.

Dessa forma o desenvolvimento de sistemas baseados em acorde encapsula sensivelmente a complexidade normalmente presente em sistemas concorrentes, uma vez que é possível abstrair o gerenciamento explícito de interações e recursos entre fluxos paralelos de execução em favor de regras declaradas explicitamente e impostas implicitamente pela implementação de acordes.

Capítulo 3

Acordes em Java

Nesse capítulo introduz-se a biblioteca para programação paralela e concorrente baseada em acordes chamada JChords. Essa biblioteca foi desenvolvida nesse trabalho como uma tentativa de introduzir a programação baseada em acordes na linguagem Java por meio de uma biblioteca externa e que, consequentemente, pudesse ser usada em um ambiente Java padrão.

É analisado o contexto geral onde JChords está inserida. São estabelecidos os requisitos da biblioteca de acordo com os objetivos que ela deve cumprir. É dada uma visão geral dos recursos e técnicas que serão utilizados para atender a esses requisitos. E também é feita uma avaliação das principais limitações que o desenho da solução impõe.

São apresentados os detalhes do desenho da biblioteca e detalhados os seus principais conceitos e como estes interagem para suportar a programação baseada em acordes por meio de biblioteca em Java.

Por fim são analisados casos de uso selecionados que demonstram o uso prático das principais funcionalidades de JChords assim como permite a análise comparativa entre sistemas baseados em acordes e seus equivalentes baseados em *threads*.

3.1 Contexto

A programação baseada em acordes é uma técnica bastante nova e que provavelmente irá amadurecer ao longo do tempo. A introdução de acordes no ecossistema Java, para ser efetiva, deve encorajar a experimentação e permitir refinamentos e evoluções. Além disso, deve prover um caminho migratório gradual, que permita a convivência harmoniosa com sistemas legados.

Propõe-se o desenvolvimento da biblioteca JChords. JChords disponibilizará os recursos necessários para programação concorrente baseada em acordes em ambientes

Java padrão. Dessa forma JChords deverá integrar harmoniosamente o ecossistema Java existente e, da maneira menos intrusiva possível, prover um caminho de transição gradual da programação sequencial ou baseada em *threads* para programação baseada em acordes.

3.1.1 Requisitos

A plataforma Java possui uma vasta base de código instalado, além de dispor de uma ampla gama de ferramentas de desenvolvimento. Dessa forma JChords foi concebida buscando introduzir o conceito de acordes em Java de maneira transparente, interagindo harmoniosamente com códigos e sistemas legados bem como ser compatível com os recursos e ferramentas de desenvolvimento atualmente disponíveis.

A Sun Microsystems é a criadora e, até hoje, principal mantenedora da linguagem Java. O conjunto de ferramentas de desenvolvimento Java (JDK) desenvolvido pela Sun é o padrão *de-facto* no que toca a forma, recursos e desempenho. Assim sendo, JChords foi desenvolvida como uma biblioteca compatível com JDK da Sun. Com isso, busca-se permitir que a programação concorrente baseada em acordes possa ser exposta a um universo mais amplo de usuários e sistemas.

Como parte do JDK, temos a cadeia de ferramentas Java que é formada pelo conjunto fundamental de ferramentas necessárias para transformar o código fonte escrito pelo desenvolvedor em uma aplicação executável. Isso inclui compiladores, otimizadores, etc. Por ser uma peça fundamental no processo de produção de software, a cadeia de ferramentas Java precisa ser estável, robusta e madura. Qualquer componente introduzido nessa cadeia precisa manter o mesmo padrão de estabilidade e robustez, sob pena de comprometer toda a cadeia.

Portanto, a introdução de ferramentas experimentais e potencialmente instáveis na cadeia de ferramentas Java é desaconselhável de modo geral e inviável em ambientes de produção. Por isso, JChords foi desenvolvida para não utilizar pré-compiladores ou etapas adicionais na cadeia de ferramentas Java, podendo ser integrado de maneira transparente com a infra-estrutura existente.

Além do conjunto fundamental de ferramentas, a plataforma Java dispõe de uma vasta biblioteca de classes padrão e de um diversificado ecossistema de bibliotecas externas, além de inúmeras ferramentas e ambientes de desenvolvimento. Deseja-se encorajar a incorporação gradual de acordes no ferramental conceitual dos desenvolvedores. Permitindo aos mesmos optar por usar acordes quando estes forem apropriados, ou recorrer a outras opções quando, por quaisquer razões técnicas ou gerenciais, o uso de acordes não for apropriado. Assim JChords deverá integrar e interagir com o ferramental Java existente de maneira transparente, incluindo o sistema tradicional de

Threads.

Em suma, para atingir seus objetivos, JChords deve cumprir os seguintes requisitos:

- Compatibilidade com Java Development Kit (JDK) da Sun Microsystems.
- Não introduzir novas dependências na cadeia de ferramentas Java.
- Coexistir com recursos, ferramentas e bibliotecas existentes.

Como já foi observado, a programação baseada em acordes é um conceito relativamente novo. Uma eventual transição da programação sequencial ou baseada em *threads* para programação baseada em acordes só será viável se for feita de forma gradual e integrada, para isso JChords precisa ser tão não-intrusiva quanto possível tanto no âmbito do código fonte quanto da infra-estrutura de desenvolvimento, encorajando o uso, experimentação e a evolução de seus conceitos.

3.1.2 Recursos

A inclusão de acordes em Java impõe uma série de demandas sintáticas e semânticas. É necessário prover meios de expressar padrões *join*, métodos assíncronos e construções auxiliares dentro da sintaxe da linguagem. É necessário também ajustar a semântica da linguagem de modo a prover os significados apropriados aos acordes.

Atender a essas demandas cumprindo os requisitos apresentados, portanto sem modificar o compilador Java, e ainda mantendo um nível de conforto apropriado a desenvolvedores Java, requer uma abordagem diferenciada que demanda ferramentas avançadas e recursos recentes. Alguns somente disponíveis a partir de Java versão 5 [Gosling et al., 2005]. Dentre os recursos utilizados em JChords, os mais significativos são:

- Anotações.
- Reflexão.
- Instrumentação.
- ASM: Biblioteca de Manipulação de *Bytecode*.

Anotações [Buckley, 2004][Sun Microsystems, 2007] é um dos recursos introduzidos na versão 5.0 da linguagem Java. As Anotações permitem associar meta-dados aos diversos elementos da linguagem como classes, métodos, campos, e até outras anotações.

As anotações por si só não afetam a semântica de um programa. Contudo as anotações podem ser processadas por meio de processadores de anotações anexados

ao compilador. Podem ainda ser armazenadas nas classes compiladas (*bytecodes*) e inspecionadas via reflexão.

Em JChords as anotações são o principal meio pelo qual os elementos sintáticos necessários a programação baseada em acordes são introduzidos no código. As anotações marcam os elementos chave do sistema e armazenam as informações complementares que permitem a atuação dos demais componentes da biblioteca.

As anotações, assim como os demais componentes do programa são acessados de duas maneiras. A primeira delas é por meio da API de reflexão integrante da biblioteca padrão Java [Sun Microsystems, 2007]. Essa API possibilita que um programa acesse, inspecione e manipule sua própria estrutura. Dessa forma, ela permite que os diversos componentes de tempo de execução de JChords obtenham acesso aos elementos do código original e possam atuar sobre eles de maneira a prover o comportamento requerido pela biblioteca.

A outra maneira utilizada para acessar os componentes do programa é a manipulação de *bytecode*. Isso é possível por meio da API de instrumentação de aplicações introduzida na biblioteca padrão do Java 5 e da biblioteca de manipulação de *bytecode* ASM [Bruneton, 2007].

A API de instrumentação permite anexar agentes à máquina virtual Java. Dentre outras funcionalidades, esses agentes são capazes de intervir no processo de leitura dos arquivos ".class" do dispositivo de armazenamento para a memória da JVM.

A biblioteca ASM por sua vez é capaz de interpretar as sequências de *bytes* que compõem os arquivos ".class", disponibilizados pela API de instrumentação e, utilizando o padrão *Visitor*[Gamma et al., 1995], possibilita a manipulação dos *bytecodes* antes que esses sejam carregados na JVM.

O conjunto dessas funcionalidades provê um *framework* efetivo para a construção do suporte a acordes em Java sem a necessidade de alterar a linguagem ou o compilador. As anotações permitem estender a sintaxe ao passo que a reflexão possibilita a criação de semânticas adaptáveis que podem ser implantadas dinamicamente via instrumentação e manipulação de *bytecodes*.

3.1.3 Limitações

Apesar de bastante versátil, a metodologia proposta apresenta algumas limitações, inerentes a decisão de implementar acordes por meio de uma biblioteca. Contudo, é esperado que os benefícios dessa abordagem superem as limitações que ela impõe.

Uma dessas limitações é a necessidade de se adequar a sintaxe padrão Java. Muito embora as anotações nos permitam anexar ao programa uma infinidade de atributos para os mais diversos usos, muito pouco pode ser feito em relação ao restante da

sintaxe. Um programa Java baseado em acordes precisa, antes de mais nada, ser um programa Java padrão válido. Ainda que a manipulação de *bytecodes* nos confira um alto grau de liberdade quanto a semântica do sistema, essa liberdade de nada vale até que o compilador tenha sido capaz de produzir classes compiladas que possam ser manipuladas. E isso só é possível a partir de códigos fonte sintaticamente válidos.

Outra limitação significativa é a impossibilidade de validar *a priori* a notação de acordes. Na maioria das linguagens de programação, uma das principais tarefas do interpretador ou compilador é detectar e reportar erros que possam existir no programa. Apesar de reduzida, a notação necessária para o suporte a acordes em Java possui regras e diretrizes que devem ser obedecidas. Como foi visto anteriormente, anotações são elementos passivos, incapazes de ativamente detectar ou reportar erros na notação de acordes. Assim, JChords não dispõe, na implementação atual, de meios para detectar esses erros até o momento em que o modulo de manipulação de *bytecodes* seja acionado, o que só ocorre já em tempo de execução.

Em essência, as limitações expostas se traduzem, de modo geral, na necessidade de construções auxiliares que seriam desnecessárias em uma implementação nativa de acordes.

3.2 Desenho

Acordes são construções compostas por um padrão *join*, que é um conjunto de mensagens ou fragmentos unidos pelo operador *join* (&), e um método ou corpo que declara quais ações estão associadas a esse padrão. Partindo da notação proposta na Seção 2.3 podemos descrever um *buffer* de inteiros utilizando acordes como mostra a Listagem 3.1.

```
1 class Buffer {
2   int get() & put(int var) {
3     return var;
4   }
5 }
```

Listagem 3.1. Pseudo-Código de buffer com acordes

Muito embora a notação proposta já mantenha muitas semelhanças a um programa Java é necessário adaptar alguns dos elementos apresentados para conformar com a sintaxe Java e para viabilizar o processamento por JChords.

Em JChords, toda classe que utiliza acordes deve ser marcada com a anotação `@Chorded`. Essa marcação permite à biblioteca identificar classes que precisam ser

manipuladas para incorporar o sistema de acordes. Dessa forma a primeira linha do exemplo passa a ser igual à Listagem 3.2.

```
1 @Chorded class Buffer ...
```

Listagem 3.2. Buffer com acordes em JChords

Por sua vez o padrão *join* apresentado não representa um trecho de código válido na linguagem Java. Por isso, em JChords, os acordes propriamente ditos são métodos convencionais, anotados com `@Join`. A anotação recebe como parâmetros as assinaturas dos fragmentos que irão compor o padrão *join* desse acorde. Esse será o método chamado quando for detectado um padrão *join* completo nesse objeto. Assim podemos re-escrever nosso exemplo da seguinte forma:

```
1 @Chorded class Buffer {
2   @Join({ "int get()", "void put(int)" })
3   int get_put(int var) {
4     return var;
5   }
6 }
```

Listagem 3.3. Buffer com join em JChords

O exemplo re-escrito já é um programa Java sintaticamente válido. Contudo invocações como `obj.put(0)` para uma instancia `obj` da classe `Buffer` seriam inválidas visto que, do ponto de vista do Java padrão a classe `Buffer` não declara um método `put(int)`.

Assim faz-se necessário declarar os métodos representativos dos fragmentos do acorde dentro da classe `Buffer` de modo que as invocações dos fragmentos de acordes sejam invocações válidas em Java padrão. Introduzir-se-a também os modificadores de acesso que estavam sendo omitidos até então. Assim re-escrevemos mais uma vez nosso exemplo como mostrado na Listagem 3.4.

```
1 @Chorded public class Buffer {
2   @Async public void put(int val) {}
3   @Sync public int get() { return Chords.result(int.class); }
4
5   @Join({ "int get()", "void put(int)" })
6   int public get_put(int var) {
7     return var;
8   }
9 }
```

Listagem 3.4. Buffer com JChords

Os métodos assíncronos são anotados com `@Async`. Eles servem para declarar os fragmentos assíncronos que poderão ser usados nos padrões *join*. Além disso, esses métodos são executados fora do fluxo de execução de sua chamada. Métodos assíncronos são os elementos chave que provêm o caráter paralelo na programação baseada em acordes.

Os métodos síncronos são anotados por `@Sync` e servem para declarar os fragmentos síncronos que poderão compor os padrões *join*. Esses métodos, isoladamente, se comportam de maneira idêntica a qualquer outro método em Java, ou seja, eles retém o fluxo de execução em que foram invocados até que sua execução tenha sido concluída.

Os métodos síncronos, ao contrário dos assíncronos que tem sempre o tipo de retorno `void`, podem retornar valor. Contudo esse deve sempre ser o valor retornado pelo corpo do acorde. Assim o valor de retorno dos métodos síncronos será sempre dado por `Chords.result()`.

Com isso temos a forma final da implementação de `Buffer` usando JChords.

JChords implementa acordes em Java pela introdução de quatro elementos básicos, cada um deles associado a uma anotação específica:

- **`@Chorded`** : Classes com Acordes
- **`@Async`** : Métodos Assíncronos
- **`@Sync`** : Métodos Síncronos
- **`@Join`** : Métodos Acordes

Um sistema baseado em acordes utilizando JChords compõe-se basicamente de classes com acordes que declaram métodos assíncronos e métodos síncronos que são empregados para compor os padrões *join* em métodos acordes.

3.2.1 Classes com Acordes

Classes com acordes são classes que suportam o uso de acordes. Adiciona-se o suporte a acordes em uma classe pelo uso da anotação `@Chorded`, e somente classes anotadas dessa forma são instrumentadas para orquestrar acordes:

```
1 @Chorded public class ChordedClass {}
```

Listagem 3.5. Classe com acordes

As demais anotações não tem nenhum efeito prático se utilizadas em classes que não tenham sido anotadas com `@Chorded`. Uma exceção são os métodos assíncronos que podem ser usados em classes com ou sem acordes, contudo, mesmo estes ficam desprovidos de mecanismos de sincronização quando usados em classes sem acordes.

Observe que nem todas as classes de um sistema concorrente baseado em acordes precisam ser instrumentadas com suporte a acordes. Via de regra somente classes que possuam pelo menos um método anotado com `@Join` deve ser anotada com `@Chorded`.

3.2.2 Métodos Assíncronos

Métodos assíncronos são métodos que são executados fora do fluxo de execução que os invocou. Da perspectiva do fluxo de execução que invoca o método assíncrono, esse método é instantâneo. Ou seja, ao contrário dos métodos normais, a chamada ao método assíncrono marca a solicitação de sua execução e não a execução propriamente dita.

Os métodos assíncronos devem ser sempre, obrigatoriamente, do tipo *void*, ou seja, não devem possuir valor de retorno. Como visto anteriormente, a ligação entre o método assíncrono e o fluxo de execução que o invoca se encerra antes da execução do método assíncrono. Com isso o método assíncrono simplesmente não tem a quem retornar um valor.

Um método poderá ser declarado assíncrono pela utilização da anotação `@Async`:

```
1 public class Example {
2   @Async public void foo() {}
3 }
```

Listagem 3.6. Método assíncrono

Vale ressaltar que os método assíncronos, por si só, podem substituir alguns dos usos mais simples de *threads*. O exemplo da Listagem 3.7 utiliza `@Async` para criar o equivalente a 5 *threads* paralelas:

```
1  public class Example {
2    @Async public void foo() {...}
3
4    public static void main(String[] args) {
5      Example example = new Example();
6      for (int index = 0; index < 5; ++index) {
7        example.foo();
8      }
9      System.out.println("fim");
10   }
11 }
```

Listagem 3.7. Exemplo de método assíncrono com JChords

Supondo que uma chamada a `foo()` requer 1 segundo para ser completada, se omitíssemos a anotação `@Async` esse programa demoraria 5 segundos antes de exibir a mensagem `fim` na tela. Contudo, se executarmos o programa com a anotação, a mensagem `fim` será exibida quase imediatamente, sendo que a execução do programa só terminará efetivamente alguns segundos depois.

A título de comparação a Listagem 3.8 mostra um programa equivalente usando *threads*.

```
1  public class Example extends Thread {
2    public void run() {...}
3
4    public static void main(String[] args) {
5      for (int index = 0; index < 5; ++index) {
6        new Example().start();
7      }
8      System.out.println("fim");
9    }
10 }
```

Listagem 3.8. Exemplo de método assíncrono com threads

Para esse exemplo trivial, observe que a sintaxe usando *threads* ou métodos assíncronos é bastante semelhante. Contudo para os casos onde houver mais de um método assíncrono, ou a necessidade de executá-los usando um único objeto a sintaxe de *threads* se torna menos prática. Isso pode ser visto comparando a Listagem 3.9 com a Listagem 3.10.

```
1  public class AsyncExample {
2    @Async public void foo() {...}
3    @Async public void bar() {...}
4
5    public static void main(String[] args) {
6      AsyncExample example = new AsyncExample();
7      for (int index = 0; index < 5; ++index) {
8        example.foo();
9        example.bar();
10     }
11   }
12 }
```

Listagem 3.9. Exemplo complexo de métodos assíncronos com JChords

```
1  public class ThreadExample {
2    public void foo() {...}
3    public void bar() {...}
4
5    public static void main(String[] args) {
6      final ThreadExample  example = new ThreadExample();
7      for (int index = 0; index < 5; ++index) {
8        new Thread() {
9          public void run() { example.foo(); }
10       }.start();
11       new Thread() {
12         public void run() { example.bar(); }
13       }.start();
14     }
15   }
16 }
```

Listagem 3.10. Exemplo complexo de métodos assíncronos com threads

Vale ressaltar que os métodos assíncronos em JChords podem ser usados de maneira independente do restante da biblioteca. Observe que os exemplos acima não utilizam nenhum dos outros elementos disponibilizados pela biblioteca, apenas `@Async`.

Isoladamente, os métodos assíncronos já proporcionam uma alternativa interessante para casos em que a sintaxe de *threads* é pouco prática. Contudo deve ser observado que, por si só, os métodos assíncronos não possuem ferramentas de sincronização e controle. Essa lacuna é preenchida pelos demais elementos da arquitetura. Em acordes, os métodos assíncronos são os elementos chave que desencadeiam o caráter paralelo do sistema.

3.2.3 Métodos Síncronos

Métodos síncronos são os meios pelos quais um sistema baseado em acordes pode recuperar os resultados de uma computação ou estabelecer os pontos de sincronização do sistema. Um método pode ser declarado síncrono pelo uso da anotação `@Sync`, como mostrado na Listagem 3.11.

```
1 @Chorded public class SyncExample {
2   @Sync public void foo() {}
3 }
```

Listagem 3.11. Método síncrono

De modo geral, é delegada aos métodos síncronos a tarefa de recuperar o resultado da execução do acorde, por isso JChords deve prover meios de materializar esse resultado dentro do corpo do método síncrono. O método estático `Chords.result(Class<T>)` provê o mecanismo para recuperar esse resultado. Ele recebe como parâmetro o objeto `Class` do tipo de resultado esperado. Via de regra, recuperar o resultado do acorde é a única operação que deve ser realizada em um método síncrono, visto que a lógica do acorde deve ficar no corpo do acorde e não nos fragmentos que o compõem.

```
1 @Chorded public class SyncExample {
2   @Sync public int bar() { return Chords.result(int.class); }
3 }
```

Listagem 3.12. Método assíncrono com valor de retorno

Os métodos síncronos possuem um caráter principalmente declarativo e em geral apenas declaram estruturas de código que serão utilizadas por outros elementos.

3.2.4 Métodos Acordes

Os métodos acordes são a construção fundamental de JChords. Eles são o meio pelo qual os acordes são declarados e definidos em um sistema baseado em acordes. Neles os padrões *join* são definidos e associados a um corpo de execução.

Os métodos acordes são anotados com `@Join`. Ao contrário das demais anotações usadas até agora, `@Join` requer um arranjo de `@strings` como parâmetro. Cada uma das `@strings` é um descritor para um dos fragmentos de acorde que compõem o padrão *join*:

```
1 @Chorded public class Buffer {
2   ...
3   @Join({ "int get()", "void put(int)" })
4   public int get_put(int var) {
5     return var;
6   }
7 }
```

Listagem 3.13. Métodos acordes

O descritor de um fragmento é uma versão estilizada da assinatura do método. Ele é composto pelo tipo de retorno, o nome e uma lista separada por virgulas e entre parênteses dos tipos de dados dos parâmetros do método. Não são necessários os nomes dos parâmetros:

```
1 "Tipo NomeDoMetodo(Tipo1,Tipo2, ..., TipoN)"
```

Listagem 3.14. Descritor de fragmento de acordes

A assinatura do método acorde deve respeitar algumas regras para que JChords possa acessar corretamente o método acorde. É importante não confundir a assinatura do acorde com a assinatura do método acorde que o representa. A assinatura do acorde é dada pela anotação `@Join` e é composta pelos descritores dos fragmentos do acorde. A assinatura do método acorde é a assinatura do método Java que será invocado quando o padrão *join* daquele acorde for detectado.

O nome do método acorde é irrelevante, pode ser qualquer nome válido para um método. Por convenção, usa-se nesse trabalho a concatenação dos nomes dos fragmentos separados por "_".

O método acorde deve obrigatoriamente ser público, caso contrário JChords não será capaz de invocá-lo via reflexão.

O tipo de retorno do método deve ser igual ao tipo de retorno do fragmento assíncrono se este existir, caso contrário deverá ser `@void`.

A lista de parâmetros do método acorde deve ser composta por todos os parâmetros de todos os fragmentos do acorde na ordem em que são declarados. Os nomes dos parâmetros não são relevantes. O *binding* será feito levando em consideração o tipo e a ordem dos parâmetros.

3.3 Aplicação

Em geral, problemas em programação paralela e concorrente se referem a conjuntos de regras de interação que regem conjuntos de entidades que interagem entre si e compartilham recursos. Os acordes permitem a representação direta das regras de interação assim como fornece formas seguras e efetivas de gerenciar os recursos compartilhados.

Assim, apresenta-se a seguir uma pequena seleção de problemas de programação paralela e concorrente assim como soluções baseadas em acordes para os mesmos, utilizando JChords. Dessa forma busca-se explorar as capacidades da biblioteca assim como introduzir as noções básicas de desenvolvimento usando JChords.

3.3.1 Produtor/Consumidor

O problema do produtor-consumidor é um problema clássico de sincronização de processos. O problema consiste em sincronizar dois grupos de processos, produtores e consumidores, por meio de um *buffer* de tamanho finito.

Os produtores produzem continuamente um determinado recurso que deve ser disponibilizado aos consumidores por meio do *buffer*. Um produtor deve ser bloqueado ao tentar adicionar um recurso ao buffer enquanto este estiver cheio.

Os consumidores por sua vez consomem continuamente os recursos disponibilizados no *buffer*. Um consumidor deve ser bloqueado ao tentar extrair um recurso do *buffer* enquanto este estiver vazio.

A implementação dos produtores e consumidores é bastante semelhante. Ambos consistem em um laço que irá continuamente produzir um recurso e armazená-lo no *buffer* ou extrair um recurso do *buffer* e consumí-lo.

Usando JChords podemos implementar a classe `Buffer` como mostrado na Listagem 3.15.

```
1  package usecases.producerconsumer.jchords;
2
3  import jchords.Chords;
4  import jchords.annotations.*;
5
6  @Chorded public class Buffer {
7    @Async public void value(int val) {}
8    @Async public void free() {}
9    @Sync public void put(int val) {}
10   @Sync public int get() { return Chords.result(int.class); }
11
12   public Buffer() {
13     for(int index = 0 ; index < Runner.BUFFER_SIZE; ++index) {
14       free();
15     }
16   }
17
18   @Join({"int get()", "void value(int)"}) public int get_value(int res) {
19     free();
20     return res;
21   }
22
23   @Join({"void put(int)", "void free()"}) public void put_free(int res) {
24     value(res);
25   }
26 }
```

Listagem 3.15. Classe `Buffer`

O acorde `get()& value(int)` é responsável pela extração de elementos. Analogamente, o acorde `put(int)& free()` é responsável pela inclusão de elementos. O fragmento `free()` representa um espaço livre no buffer, ao passo que o fragmento `value(int)` representa um espaço ocupado.

Instâncias de `Buffer` são inicializadas com tantas chamadas a `free()` quanto o número de elementos que o *buffer* deve comportar. A qualquer tempo, haverá no *buffer* tantas chamadas pendentes ao fragmento `free()` quanto houver espaços livres e tantas chamadas pendentes ao fragmento `value(int)` quanto houver elementos armazenados, sendo que a soma de chamadas pendentes desses fragmentos será sempre igual à capacidade do *buffer*. Isso porque o acorde que consome as chamadas a `free()` sempre invoca `value(int)` e vice-versa.

Sempre que um consumidor deseja extrair elementos do *buffer* ele invoca o fragmento `get()`. Existindo uma chamada pendente à `value(int)`, então o acorde `get()& value(int)` será executado. Caso contrário a chamada ao fragmento `get()`, que é um fragmento síncrono, irá bloquear o fluxo de execução até que `value(int)` seja chamado.

A execução do acorde `get()& value(int)` consome uma chamada pendente a `get()` e uma a `value(int)`. Esse acorde invoca `free()` que disponibiliza um espaço livre no *buffer* e retorna ao código que invocou o fragmento `get()` o valor passado como parâmetro ao fragmento `value(int)`.

Analogamente, sempre que um produtor deseja incluir elementos no *buffer* ele invoca o fragmento `put(int)`. Se houver uma chamada pendente à `free()` o acorde `put(int)& free()` é executado. Caso contrário `put(int)` bloqueia o fluxo de execução até uma chamada a `free()`.

A execução do acorde `put(int)& free()` consome, pelo padrão *join*, uma chamada pendente a `put(int)` e uma a `free()`. Por fim, invoca o fragmento `value(int)` usando o parâmetro passado ao fragmento `put(int)`.

A título de comparação a Listagem 3.16 mostra a classe `Buffer` implementada em Java usando *threads*.

```
1  package usecases.producerconsumer.java;
2
3  public class Buffer {
4    private int count = 0;
5    private int[] resources = new int[Runner.BUFFER_SIZE];
6
7    public synchronized void put(int res) throws InterruptedException {
8      while (count == resources.length) {
9        wait();
10     }
```

```
11      resources[count++] = res;
12      notifyAll();
13    }
14
15    public synchronized int get() throws InterruptedException {
16      while (count == 0) {
17        wait();
18      }
19      int res = resources[--count];
20      notifyAll();
21      return res;
22    }
23  }
```

Listagem 3.16. Classe `Buffer`

Vale observar que a implementação usando acordes abstrai o desenvolvedor do gerenciamento de *threads*, notadamente o trio `wait()/notify()/notifyAll()`, e permite focar nas regras do problema. De fato, os acordes são, em essência, traduções direta das regras do problema:

- `get()& value(int)`: Elementos só podem ser extraídos do *buffer* se houver elementos no buffer, caso contrário bloqueia.
- `put(int)& free()`: Elementos só podem ser incluídos no *buffer* se houver espaços livres no buffer, caso contrário bloqueia.

Além disso os recursos compartilhados, no caso os recursos sendo produzidos e consumidos, ficam verdadeiramente protegidos, estando acessíveis somente às entidades apropriadas, nos momentos e escopos apropriados.

3.3.2 Jantar dos Filósofos

O jantar dos filósofos é um problema clássico de concorrência. O problema consiste em cinco filósofos sentados ao redor de uma mesa. Sobre a mesa existem cinco garfos de modo que cada filósofo dispõe de dois garfos, cada garfo compartilhado com um de seus vizinhos. A vida de um filósofo se resume a pensar e comer macarrão. Cada filósofo pensa por algum tempo, pega um garfo, depois o outro, come macarrão, devolve os garfos e volta a pensar.

Apresentam-se duas soluções para o problema do jantar dos filósofos. A primeira é a solução do garçom que consiste em criar um elemento central de controle que irá

gerenciar os recursos. A segunda é uma solução totalmente distribuída proposta por Chandy & Misra [1984], chamada solução higiênica.

3.3.2.1 A Solução do Garçom

A solução do garçom consiste em criar a figura de um garçom. Todos os filósofos pedem permissão ao garçom antes de pegar os garfos e informam ao mesmo quando devolvem os garfos. O garçom, tendo uma visão global do processo, pode coordenar os filósofos de maneira adequada.

A estratégia de gerenciamento adotada aqui para o garçom foi a de limitar o número máximo de filósofos disputando os recursos a quatro filósofos, ou seja, o total de filósofos menos um.

Dessa forma, implementa-se o garçom por meio da classe `Waiter` apresentada na Listagem 3.17.

```
1  package usecases.diningphilosophers.waiter.jchords;
2
3  import jchords.annotations.*;
4
5  @Chorded public class Waiter {
6    @Sync public void enter() {}
7    @Async public void leave() {}
8
9    public Waiter() {
10     for (int index = 0; index < Runner.SEATS - 1; ++index) {
11       leave();
12     }
13   }
14
15   @Join({"void enter()", "void leave()"}) public void enter_leave() {}
16 }
```

Listagem 3.17. Classe `Waiter`

As instâncias da classe `Waiter` são inicializadas com quatro chamadas ao fragmento `leave()`, ou seja, o número de filósofos menos um. O fragmento `leave()` representa uma permissão para um filósofo pegar os garfos.

O acorde `enter()& leave()` simplesmente irá bloquear qualquer chamada a `enter()` até que haja uma chamada pendente a `leave()`, dessa forma somente os quatro primeiros filósofos que tentarem comer serão autorizados a tentar. O último filósofo será obrigado a esperar até que um dos demais tenha terminado.

Novamente a título de comparação a Listagem 3.18 mostra a classe `Waiter` implementada em Java com *threads*.

```
1  package usecases.diningphilosophers.waiter.java;
2
3  public class Waiter {
4    private int count = Runner.SEATS - 1;
5
6    public synchronized void enter() throws InterruptedException {
7      while(count > 0) {
8        wait();
9      }
10     --count;
11   }
12
13   public synchronized void leave() {
14     ++count;
15     notifyAll();
16   }
17 }
```

Listagem 3.18. Classe `Waiter`

3.3.2.2 A Solução Higiênica

A solução higiênica para o problema do jantar dos filósofos foi proposta por Chandy & Misra [1984]. Nessa solução, cada garfo pode estar sujo ou limpo. Sempre que um filósofo usa o garfo esse fica sujo. Um filósofo sempre limpa o garfo antes de emprestá-lo a um vizinho, daí o nome solução higiênica. Sempre que tem fome um filósofo pede emprestado aos vizinhos os garfos que não possua. Um filósofo não atende a pedidos de empréstimo de garfos enquanto estiver comendo.

Em relação a solução do garçom, a solução higiênica apresenta a significativa vantagem de não depender de uma figura central de controle, consequentemente ela pode ser aplicada em ambientes distribuídos. Além disso, exceto pela inicialização, ela é simétrica, ou seja, assim como a solução do garçom, todos os filósofos são instâncias da mesma classe e compartilham a mesma implementação.

Assim, implementamos os filósofos por meio da classe `Phil` apresentada na Listagem 3.19:

```
1  package usecases.diningphilosophers.higienic.jchords;
2
```

```
3  import static jchords.util.Utils.delay;
4  import jchords.annotations.*;
5
6  @Chorded public class Phil {
7    private static int seed = 0;
8    public final int id = ++seed;
9
10   @Async public void requires(Phil phil) {}
11   @Async public void clean(Fork fork) {}
12   @Async public void dirty(Fork fork) {}
13   @Async public void request(Phil phil) {}
14   @Async public void hungry() {}
15   @Sync public void eating() {}
16
17   @Join({"void eating()", "void hungry()",
18     "void clean(usecases.diningphilosophers.higienic.jchords.Fork)",
19     "void clean(usecases.diningphilosophers.higienic.jchords.Fork)"})
20   public void eat_hungry(Fork left, Fork right) {
21     left.grab();
22     right.grab();
23     eat();
24     left.release();
25     right.release();
26     dirty(left);
27     dirty(right);
28   }
29
30   @Join({"void hungry()",
31     "void requires(usecases.diningphilosophers.higienic.jchords.Phil)"})
32   public void hungry_requires(Phil phil) {
33     hungry();
34     phil.request(this);
35   }
36
37   @Join({"void request(usecases.diningphilosophers.higienic.jchords.Phil)",
38     "void dirty(usecases.diningphilosophers.higienic.jchords.Fork)"})
39   public void request_dirty(Phil phil, Fork fork) {
40     requires(phil);
41     phil.clean(fork);
42   }
```

```
43
44   @Async public void execute() {
45     while (true) {
46       think();
47       hungry();
48       eating();
49     }
50   }
51
52   private void eat() {
53     delay(5000, "Filósofo " + id + " comendo");
54   }
55
56   private void think() {
57     delay(1000, "Filósofo " + id + " pensando");
58   }
59 }
```

Listagem 3.19. Classe `Phil`

O ciclo de vida do filósofo consiste em invocar os métodos ou fragmentos `think()`, `hungry()` e `eating()`. Desses, o fragmento assíncrono `hungry()` é responsável por estimular o filósofo a iniciar as ações necessárias para que este possa comer. Por sua vez o fragmento síncrono `eating()` irá reter o fluxo de execução do filósofo até que ele tenha todas as condições necessárias para comer.

O acorde `eating()& hungry()& clean(Fork)& clean(Fork)` será executado quando o filósofo dispuser de todas as condições necessárias para comer. Ou seja, o filósofo deve estar com fome (`hungry()`), deve estar aguardando para comer (`eating()`) e deve ter dois garfos disponíveis (`clean(Fork)` duas vezes). Ao executar esse acorde os dois garfos usados ficam sujos, o que será sinalizado pela invocação dupla do fragmento `dirty(Fork)`.

Vale ressaltar que a solução de Chandy & Misra [1984] não requer a disponibilidade de garfos limpos. Essa restrição foi usada apenas para simplificar o exemplo. Uma implementação completa deve conter acordes para tratar casos onde haja um ou dois garfos sujos.

Uma vez que o filósofo esteja com fome, mas não possua algum dos garfos, o acorde `hungry()& requires(Phil)` emitirá solicitações aos filósofos para os quais este tenha emprestado garfos. Uma solicitação é feita enviando o fragmento `request(Phil)` para o filósofo recebido como parâmero ao fragmento `requires(Phil)`. Além disso, é necessário reinvocar o fragmento `hungry()` que foi consumido por esse acorde de modo que ele fique

disponível para compor mais padrões *join*.

Por fim, o acorde `request(Phil)& dirty(Fork)` é responsável por enviar o garfo a algum filósofo solicitante, se o garfo estiver disponível. Esse acorde também registra para qual filósofo o garfo está sendo emprestado invocando o fragmento assíncrono `requires(Phil)`. Desse modo é possível solicitar o garfo de volta quando necessário. Além disso o garfo passa a estar limpo antes de ser enviado. Isso é feito pelo fragmento `clean(Fork)` enviado ao filósofo requisitante.

Os garfos sujos e limpos criam uma ordem de precedência para o empréstimo dos garfos. Isso evita que um filósofo empreste um garfo que ele tenha acabado de obter, evitando assim que ele fique indefinidamente com fome [Chandy & Misra, 1984].

3.3.3 O Problema do Papai Noel

O problema do Papai Noel foi proposto por Trono [1994] como um exercício não-trivial em programação concorrente. O problema é composto por dez elfos, nove renas e pelo Papai Noel. A vida do Papai Noel se resume a dormir até que ele seja acordado pelas nove renas ou por um grupo de três elfos. Caso seja acordado pelas renas o Papai Noel amarra-as ao trenó, distribui brinquedos, desamarra-as e volta a dormir. Caso seja acordado pelos duendes, o Papai Noel discute projetos de brinquedos com os elfos e volta a dormir. A vida de uma rena se resume a entregar presentes e tirar férias até o próximo ano. A vida de um elfo se resume a fabricar brinquedos e se reunir com o Papai Noel.

A solução apresentada se baseia em Benton [2003]. Inicialmente definimos um grupo, ou `NWay` na solução original, representado pela classe com acorde `Group`.

```
1  package usecases.santaclaus.jchords;
2
3  import jchords.annotations.*;
4
5  @Chorded public class Group {
6    @Sync public void entry() {}
7    @Async public void tokens(int n) {}
8    @Sync public void waitt() {}
9    @Async public void allgone() {}
10
11   public void accept(int n) {
12     tokens(n);
13     waitt();
14   }
15
```

```
16   @Join({"void entry()", "void tokens(int)"})
17   public void entry_tokens(int n) {
18     if (--n == 0) {
19       allgone();
20     } else {
21       tokens(n);
22     }
23   }
24
25   @Join({"void waitt()", "void allgone()"}) public void wait_allgone() {}
26 }
```

Listagem 3.20. Classe `Group`

Um grupo aceita o método `accept(int n)` que define que este grupo estará completo quanto tiver `n` elementos inscritos. Esse método retém o fluxo de execução até que o grupo esteja completo. Um elemento se inscreve no grupo chamando o método `entry` que por sua vez também retém o fluxo de execução até que o grupo esteja completo. O grupo é essencialmente uma barreira que irá reter um grupo de elementos até que todos eles estejam prontos.

```
1  package usecases.santaclaus.jchords;
2
3  import jchords.annotations.*;
4
5  @Chorded public class SantaClaus {
6    public static final int REIN_TEAM = 9;
7    public static final int ELF_TEAM = 3;
8    public static SantaClaus INSTANCE = new SantaClaus();
9
10   public final Group harness = new Group();
11   public final Group unharness = new Group();
12   public final Group roomin = new Group();
13   public final Group roomout = new Group();
14
15   private SantaClaus() {
16     elves(0);
17     reins(0);
18     elvesfirst();
19     sleep();
20   }
```

```
21
22   @Async public void sleep() {
23     System.out.println("Papai Noel dormindo");
24   }
25   @Async public void elf() {}
26   @Async public void elves(int e) {}
27   @Async public void elvesready() {}
28   @Async public void elvesfirst() {}
29   @Async public void rein() {}
30   @Async public void reins(int i) {}
31   @Async public void reinsready() {}
32   @Async public void reinsfirst() {}
33
34   @Join({"void reinsfirst()", "void elvesfirst()"})
35   public void reinfirst_elvesfirst() {}
36
37   @Join({"void elf()", "void elves(int)"}) public void elf_elves(int e) {
38     if (++e == ELF_TEAM) {
39       elvesready();
40     } else {
41       elves(e);
42     }
43   }
44
45   @Join({"void rein()", "void reins(int)"}) public void rein_reins(int r) {
46     if (++r == REIN_TEAM) {
47       reinsfirst();
48       reinsready();
49     } else {
50       reins(r);
51     }
52   }
53
54   @Join({"void sleep()", "void reinsready()"})
55   public void sleep_reinsready() {
56     harness.accept(REIN_TEAM);
57     elvesfirst();
58     reins(0);
59     delivering();
60     unharness.accept(REIN_TEAM);
```

```
61      sleep();
62    }
63
64    @Join({"void sleep()", "void elvesready()", "void elvesfirst()"})
65    public void sleep_elvesready() {
66      elvesfirst();
67      roomin.accept(ELF_TEAM);
68      elves(0);
69      consulting();
70      roomout.accept(ELF_TEAM);
71      sleep();
72    }
73
74    private void consulting() {
75      System.out.println("Papai Noel consultando");
76    }
77
78    private void delivering() {
79      System.out.println("Papai Noel entregando brinquedos");
80    }
81  }
```

Listagem 3.21. Classe `SantaClaus`

O Papai Noel possui quatro grupos. Renas amarradas, renas desamarradas, elfos entrando na reunião e elfos saindo da reunião. Além disso, os parâmetros dos fragmentos assíncronos `reins(int r)` e `elves(int r)` mantém controle de quantos elfos e renas, respectivamente, estão aguardando a atenção do Papai Noel.

Sempre que uma nova rena volta de férias ela invoca o fragmento `rein()` e então invoca o fragmento síncrono `entry()` do grupo de renas amarradas, o que retém sua execução até que o grupo de renas amarradas esteja completo. Por sua vez sempre que um elfo deseja se reunir com o Papai Noel ele invoca o fragmento assíncrono `elf()` e então invoca o fragmento síncrono `entry()` do grupo de elfos entrando na reunião, o que também retém seu fluxo de execução até que o grupo elfos entrando na reunião esteja completo.

Invocações aos fragmentos `rein()` e `elf()` causam a execução dos acordes `rein_reins(int r)` e `elf_elves(int e)` que atualizam o número de renas e elfos aguardando a atenção do Papai Noel. Quando o número de renas ou elfos atingem o valor desejado, isso causa a invocação dos fragmentos `reinsready()` ou `elvesready()`. Estes por sua vez podem causar a execução dos acordes `sleep_reinsready()` ou `sleep_elvesready`

`()` que executam uma viagem de entrega de brinquedos ou uma reunião com os elfos, respectivamente.

Os fragmentos `reinsfirst()` e `elvesfirst()`, e o acorde `reinfirst_elvesfirst()` nos permitem priorizar as renas, garantindo que o acorde `sleep_elvesready()` só será executado se o fragmento `elvesfirst()` estiver presente.

3.4 Conclusões

JChords disponibiliza a programação concorrente baseada em acordes em Java por meio de uma biblioteca externa. Assim é possível integrar os acordes de maneira gradual ao ferramental Java existente e incentivar a experimentação e evolução.

Para isso foram desenvolvidos mecanismos que exploram os novos recursos do Java 5 para viabilizar a descrição e implementação de soluções baseadas em acordes sem demandar alterações na linguagem.

Apesar das limitações que essa abordagem impõe, JChords se mostrou uma ferramenta efetiva na solução de um conjunto de problemas de sincronização, concorrência e paralelismo. Permitindo expressar soluções de maneira clara e direta.

A biblioteca JChords não é capaz de resolver por si só todos os problemas envolvidos na criação de sistemas paralelos. Sua principal contribuição está em permitir que sistemas paralelos e concorrentes sejam especificados em um maior nível de abstração de uma forma mais simples, efetiva e menos propensa a erros. Toda solução em JChords pode ser mapeada para uma solução equivalente usando as construções usuais em Java.

JChords busca liberar o desenvolvedor do gerenciamento de *threads*, permitindo que o foco seja mantido nas demandas do sistema em si, e não nas possíveis interações adversas entre fluxos paralelos de execução.

Capítulo 4

Implementação

Nesse capítulo detalha-se a implementação de JChords. Apresenta-se uma visão geral dos componentes básicos da biblioteca. São exploradas as interações entre esses componentes e como eles colaboram para especificar, transformar e executar programas baseados em acordes a partir de sistemas Java convencionais.

Posteriormente cada um desses componentes é analisado em detalhes. Sendo apresentada a estrutura interna de cada um deles assim como os detalhes de implementação e decisões de projeto mais importantes.

4.1 Camadas de Atuação de JChords

A biblioteca JChords é composta por três camadas de atuação. A camada de desenvolvimento, a camada de transformação e a camada de execução. Cada uma dessas camadas atua em um momento específico do ciclo de vida de um sistema Java e disponibiliza um conjunto de recursos para as camadas seguintes. A Figura 4.1 ilustra essa estrutura.

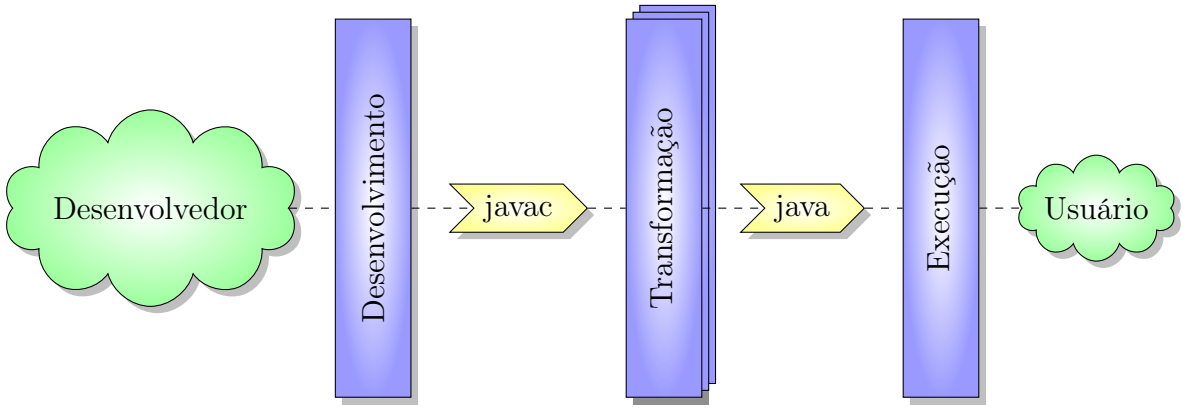

Figura 4.1. Camadas de atuação

A camada de desenvolvimento atua durante a codificação do sistema concorrente baseado em acordes. Nela são disponibilizadas as anotações que permitem ao desenvolvedor anexar as meta-informações necessárias ao código. Com isso permitindo a codificação dos acordes e estruturas de apoio junto ao código fonte Java convencional.

A camada de transformação atua junto à máquina virtual Java no momento em que esta carrega as classes compiladas em *bytecode* do dispositivo de armazenamento e antes do início da execução do sistema. Essa camada é formada por uma sequência encadeada de filtros ou agentes de transformação. Cada um desses filtros monitora o fluxo de *bytecodes* em busca das anotações disponibilizadas pela camada de desenvolvimento, manipulando as sequências de instruções quando necessário. Dessa forma essa camada instrumenta as classes com os elementos necessários para suportar acordes durante a execução do sistema.

Por fim, a camada de execução provê a infraestrutura que irá suportar o sistema baseado em acordes durante sua execução. Nessa camada o subsistema de casamento de padrões monitora as chamadas aos fragmentos síncronos e assíncronos, coordena a formação e sincronização dos padrões *join* resultantes e invoca a execução dos acordes.

A associação entre anotações em código e instrumentação de classes via manipulação de *bytecodes* usada em JChords é uma técnica bastante poderosa para adição de novas funcionalidades à linguagem sem a necessidade de modificar o compilador ou a própria linguagem. Essa técnica permite à biblioteca reusar a sintaxe convencional da linguagem, modificando a semântica de certas construções em condições específicas do código compilado.

4.2 Camada de Desenvolvimento

A camada de desenvolvimento é a parte do sistema que fica visível ao desenvolvedor. Ela provê as construções usadas durante a elaboração dos arquivos fonte que descrevem o sistema concorrente baseado em acordes. Essa camada é composta primariamente por anotações. Essas anotações permitem anexar ao código fonte, meta-informações que especificam as classes, fragmentos e acordes que compõem o sistema baseado em acordes.

Em Java, uma anotação é especificada utilizando uma sintaxe semelhante a usada para especificar interfaces:

```
1 public @interface AnnotationName {}
```

Listagem 4.1. Anotações

Um declaração de anotação, assim como diversas outras construções em Java, podem ser anotadas. A biblioteca padrão Java provê as anotações `@Target` e `@Retention` que permitem especificar, respectivamente, o conjunto de elementos aos quais uma anotação se aplica e até que ponto da vida útil do sistema essa anotação deve ser preservada.

Assim, a anotação `@Chorded` é definida da seguinte forma:

```
1 package jchords.annotations;
2
3 import java.lang.annotation.*;
4
5 @Retention(RetentionPolicy.RUNTIME)
6 @Target(ElementType.TYPE)
7 public @interface Chorded {}
```

Listagem 4.2. Anotação `@Chorded`

`@Chorded` é aplicável a tipos, ou seja, classes, interfaces, enumerações e anotações. Contudo JChords somente irá instrumentar classes anotados com `@Chorded`, os demais tipos serão ignorados. Além disso `@Chorded` é uma anotação que deve ser preservada até o tempo de execução, ou seja, ela irá compor os *bytecodes* dos arquivos compilados e poderá ser inspecionada em tempo de execução por meio de reflexão. Consequentemente, essa anotação estará disponível para a camada de transformação em tempo de carga.

As anotações `@Async` e `@Sync` são definidas de maneira análoga. Ambas também são disponibilizadas em tempo de execução. Porém elas são aplicáveis somente a métodos.

Além das políticas de aplicação e retenção, uma anotação pode possuir propriedades. Essas propriedades permitem ao desenvolvedor detalhar e complementar a anotação com trechos específicos de dados. A anotação `@Join` utiliza esse mecanismo para permitir que seja especificado o arranjo de descritores de fragmentos de acordes que irão compor o padrão *join* de um método acorde:

```
1 package jchords.annotations;
2
3 import java.lang.annotation.*;
4
5 @Retention(RetentionPolicy.RUNTIME)
6 @Target(ElementType.METHOD)
7 public @interface Join {
8   String[] value();
9 }
```

Listagem 4.3. Anotação `@Join`

Em `@Join`, a propriedade `value()` contém o arranjo de *strings* que é usado para especificar o padrão *join* em tempo de desenvolvimento. O valor dessa propriedade será armazenado nas classes compiladas e será usado para alimentar o subsistema de casamento de padrões.

As anotações são, por definição, elementos passivos no código. Não há uma ação ou execução associado a uma anotação. Seu objetivo é prover subsídios para outras ferramentas ou componentes do sistema. Da mesma forma as anotações disponibilizadas pela camada de desenvolvimento irão apenas fornecer subsídios para as demais camadas de atuação de JChords.

4.3 Camada de Transformação

A camada de transformação é responsável por instrumentar as classes com acordes, provendo a infraestrutura necessária para seu funcionamento. Essa camada atua no momento entre a carga das classes do dispositivo de armazenamento e o início da execução do sistema. Essa camada se sustenta sobre duas outras tecnologias. A primeira é a API de instrumentação Java, disponibilizada pela Sun a partir da versão 5 do Java. A outra é a biblioteca de manipulação de bytecodes ASM da ObjectWeb [Bruneton, 2007].

A API de instrumentação Java possibilita anexar agentes de transformação à JVM. Esses agentes devem disponibilizar o método estático `premain` que, como o próprio nome indica, é invocado antes do método estático `main`, que inicia a execução de um sistema Java. Mais importante ele é executado antes que as demais classes do sistema sejam carregadas pela JVM, o que permite intervir nesse processo de carga.

A interface `ClassFileTransformer` é uma das interfaces disponibilizadas pela API de instrumentação. Ela permite inspecionar e transformar um arranjo de *bytecodes* que integram as classes. A classe `Transformer` descrita na Listagem 4.4 implementa essa interface.

```
1 package jchords.agents;
2
3 import static java.lang.Boolean.parseBoolean;
4 import static java.lang.System.getProperty;
5 import static org.objectweb.asm.ClassReader.*;
6 import static org.objectweb.asm.ClassWriter.COMPUTE_FRAMES;
7 import java.io.PrintWriter;
```

```
8  import java.lang.instrument.*;
9  import java.security.ProtectionDomain;
10 import org.objectweb.asm.*;
11 import org.objectweb.asm.util.*;
12
13 public class Transformer implements ClassFileTransformer {
14   public static final boolean SHOW_BYTECODE = parseBoolean(
15     getProperty("jchords.showbytecode")
16   );
17
18   public byte[] transform(ClassLoader loader, String name, Class cls,
19     ProtectionDomain domain, byte[] bytes) {
20     ClassReader reader = new ClassReader(bytes);
21     ClassWriter writer = new ClassWriter(reader, COMPUTE_FRAMES);
22     try {
23       reader.accept(adapt(writer), SKIP_FRAMES);
24     } catch (Throwable ex) {
25       ex.printStackTrace();
26     }
27     return writer.toByteArray();
28   }
29
30   private ClassVisitor adapt(ClassVisitor cv) {
31     if (SHOW_BYTECODE) {
32       cv = new TraceClassVisitor(cv, new PrintWriter(System.out));
33     }
34     cv = new CheckClassAdapter(cv);
35     cv = new AsyncTransformer(cv);
36     cv = new JoinTransformer(cv);
37     return cv;
38   }
39
40   public static void premain(String args, Instrumentation instrumentation) {
41     instrumentation.addTransformer(new Transformer());
42   }
43 }
```

Listagem 4.4. Classe `Transformer`

A classe `Transformer` responde por duas funções básicas. A primeira é realizada pelo método estático `premain` e consiste em anexar uma instância de si próprio ao

conjunto de `ClassFileTransformer` da API de instrumentação. A outra é aplicar a cadeia de agentes de transformação aos *bytecodes* das classes.

A biblioteca ASM utiliza o padrão *visitor* e o padrão *adapter* para permitir a inspeção e manipulação dos *bytecodes* de uma classe. A Figura 4.2 ilustra a estrutura básica de uma solução utilizando ASM.

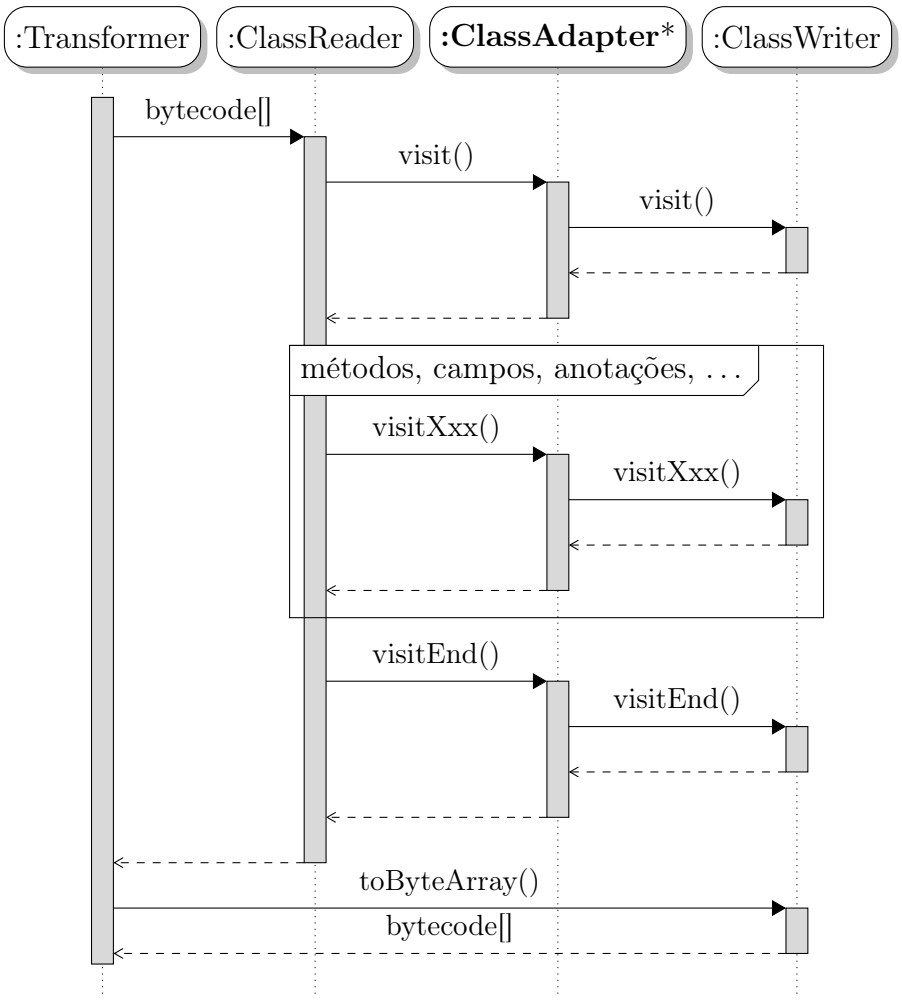

Figura 4.2. Estrutura de uma solução em ASM

A classe `ClassReader` recebe como parâmetros um arranjo de *bytes* contendo os *bytecodes* que definem a classe, é uma implementação da interface `ClassVisitor`. `ClassReader` irá interpretar o arranjo de *bytes* e invocar em sequência os métodos de visita, do tipo `visitXxx(...)`, de `ClassVisitor`. Cada método de visita reflete alguma estrutura ou comando encontrado nos *bytecodes* como `visitMethod(...)` para métodos, `visitField(...)` para campos de dados ou `visitAnnotation(...)` para anotações.

Analogamente, temos a classe `ClassWriter`, que implementa `ClassVisitor` e realiza a função oposta de `ClassReader`. `ClassWriter` recebe uma sequência de chamadas aos

seus métodos `visitXxx(...)` e gera um arranjo de *bytes* que definem uma classe na máquina virtual Java.

Por fim, ASM provê a classe auxiliar `ClassAdapter` que simplesmente redireciona as chamadas dos métodos de visita a outra realização de `ClassVisitor`. Com isso, podemos criar um encadeamento de filtros, começando pelo `ClassReader`, passando por várias subclasses de `ClassAdapter` e terminando em `ClassWriter`. Cada filtro, sobrescreve um conjunto de métodos de visita de interesse e repassa as chamadas ao próximo filtro da cadeia.

Utilizando essa técnica, JChords define os filtros, ou agentes de transformação, `AsyncTransformer JoinTransformer`. Sendo que `JoinTransformer` por sua vez é composto pelos agentes `ManagerMethod`, `SyncMethod` e `AsyncMethod`.

4.3.1 AsyncTransformer

O agente de transformação `AsyncTransformer` é responsável por transformar os métodos anotados com `@Async`, que em Java convencional são síncronos, em métodos assíncronos, ou seja, métodos que são executados fora do fluxo de execução que os invocou.

Vale ressaltar que `AsyncTransformer` serve unicamente para proporcionar a execução concorrente de um método. Ela não realiza nenhuma transformação relacionada ao casamento de padrões *join* ou invocação de acordes. Uma das vantagens da arquitetura de encadeamento de agentes de transformação é que pode-se criar agentes completamente independentes, que se complementam em determinado contexto, mas que podem ser utilizados separadamente.

Para transformar métodos convencionais em métodos assíncronos, são necessárias duas operações. A primeira é renomear o método original e torná-lo privado. A segunda opção é criar um novo método com a mesma assinatura do método original e inserir o código de invocação assíncrona, que utiliza uma nova *threads* para invocar o método original renomeado. Assim, o exemplo do código na Listagem 4.5.

```
1 public class Example {
2   @Async public void foo(int x, String y) {
3     // Corpo do método
4   }
5 }
```

Listagem 4.5. Método assíncrono original

será transformado em algo semelhante ao código da Listagem 4.6.

```
1 public class Example {
2   private void Async$foo(int x, String y) {
```

```
3       // Corpo do método
4     }
5     public void foo(final int x, final String y) {
6       new Thread() {
7         public void run() {
8           Async$foo(x,y);
9         }
10      }.start();
11    }
12  }
```

Listagem 4.6. Método assíncrono transformado

Na prática JChords, delega a chamada assíncrona à classe `@AsyncCall`. Essa classe realiza algumas adaptações internas, devido ao modo como classes anônimas são tratadas na JVM, e invoca o método original via reflexão. Contudo o efeito final é equivalente ao da Listagem 4.6.

4.3.2 JoinTransformer

O `JoinTransformer` por si só, é responsável unicamente por identificar as classes anotadas com `@Chorded`. Quando uma classe com acordes é detectada, três outros agentes de transformação são acionados para efetivamente instrumentar a classe. São eles `ManagerMethod`, `SyncMethod` e `AsyncMethod`.

O agente `ManagerMethod` é responsável por adicionar à classe os mecanismos necessários para instanciar e acessar o gerenciador de acordes. O gerenciador de acordes é a peça central da camada de execução de JChords. Além disso, esse agente é responsável por coletar todos os padrões *join* declarados pela anotação `@Join` e registrá-los junto ao gerenciador de acordes.

`AsyncMethod` é responsável por instrumentar os métodos assíncronos de modo a notificar o gerenciador de acordes sempre que esses métodos forem invocados. É importante ressaltar que `AsyncMethod` exerce uma função complementar porém bastante distinta do `AsyncTransformer` visto anteriormente. O trabalho do `AsyncTransformer` é prover meios para que um dado método seja executado concorrentemente ao fluxo principal. Por sua vez, `AsyncMethod` é responsável por garantir que esses métodos integrem a arquitetura de acordes e componham o casamento de padrões *join*.

Por fim o agente `AsyncMethod` instrumenta os métodos síncronos para notificar o gerenciador de acordes e para buscar junto a esse o resultado da execução do acorde. Em essência esse agente substitui a chamada a `Chords.result()` pela chamada ao

gerenciador de acordes. Essa chamada é inserida no fim do método no caso de ele não retornar valores.

4.3.3 Transformação

O produto final da camada de transformação é uma classe Java devidamente instrumentada para operar com acordes. Cada um dos agentes trata de uma transformação específica, e em conjunto eles produzem uma classe baseada em acordes que será carregada e executada em uma máquina virtual Java padrão. No fim do processo completo de transformação, a classe `Buffer` mostrada na Listagem 4.7.

```
1  package usecases.simplebuffer.jchords;
2
3  import jchords.Chords;
4  import jchords.annotations.*;
5
6  @Chorded public class Buffer {
7    @Async public void put(int var) {}
8    @Sync public int get() { return Chords.result(int.class); }
9    @Join({"int get()", "void put(int)"}) public int get_put(int var) {
10     return var;
11   }
12 }
```

Listagem 4.7. Classe `Buffer` original

seria transformada de modo a produzir uma classe semelhante a da Listagem 4.8.

```
1  package usecases.simplebuffer.jchords;
2
3  import jchords.*;
4  import jchords.annotations.*;
5  import jchords.annotations.Join;
6
7  @Chorded class BufferClass {
8    public BufferClass() {
9      $initChordManager$();
10   }
11
12   @Async public void Async$put(int value) {
13     $ChordManager$.addAsync("void put(int)", value);
14   }
```

```
15
16   public void put(int value) {
17     AsyncCall.invoke(BufferClass.class, "Async$put",
18       new Class<?>[] {int.class}, this, new Object[] {value});
19   }
20
21   @Sync public int get() {
22     return (Integer)$ChordManager$.addSync("int get()");
23   }
24
25   @Join({ "get()", "put(int)"}) public int get_put(int value) {
26     return value;
27   }
28
29   protected Chords $ChordManager$;
30
31   protected void $initChordManager$() {
32     $ChordManager$ = new Chords(this,
33       new JoinDescriptor("int get_put(int)", "int get()","void put(int)")
34     );
35   }
36 }
```

Listagem 4.8. Classe `Buffer` transformada

É importante observar que não é possível expressar uma tradução exata do processo de transformação em um código na linguagem Java, tendo em vista que algumas das manipulações de *bytecode* realizadas não são passíveis de serem expressas diretamente na linguagem Java.

4.4 Camada de Execução

A camada de execução, atua durante a execução do sistema baseado em acordes. Ela é responsável por prover a infraestrutura que será utilizada pelas classes instrumentadas na camada de transformação. Ela provê os componentes que irão monitorar e registrar a invocação de fragmentos dos acordes, detectar os padrões *join*, coordenar, sincronizar e executar os acordes.

Essa camada é composta de um único componente principal que encapsula um pequeno conjunto de componentes auxiliares. O componente principal dessa camada é o gerenciador de acordes, implementado pela classe `Chords`. Essa é a mesma classe que

provê o método estático `Chords.result()`, usado para obter o resultado de um acorde. A camada de transformação garante que cada instância de uma classe com acordes possua um membro protegido do tipo `Chords`.

4.4.1 Chords

A principal tarefa do gerenciador de acordes é realizar o casamento de padrões *join*.

Como vimos na camada de transformação, ao ser inicializado, o gerenciador de acordes recebe um conjunto de instâncias da classe `JoinDescriptor`. Cada instância de `JoinDescriptor` é composta pelos descritores de fragmentos atribuídos a um dado método acorde pela anotação `@Join`, precedido pelo descritor do próprio método acorde:

```
1  new Chords(this,
2    new JoinDescriptor("int get_put(int)", "int get()","void put(int)")
3  );
```

Listagem 4.9. Inicialização do gerenciador de acordes

JChords usa um sistema de casamento de padrões em árvore baseado no sistema proposto por Itzstein [2005]. Dessa forma, ao ser inicializado, o gerenciador de acordes irá utilizar o conjunto de `JoinDescriptor` para construir a árvore de casamento de padrões que será usada pela instancia da classe com acordes.

A árvore de casamento de padrões, a rigor, não é uma árvore, mas um grafo dirigido. Esse grafo possui três níveis, sendo que a raiz desse grafo e o próprio gerenciador de acordes.

O segundo nível é formado pelo conjunto de padrões *join*. Cada acorde na classe dá origem a um padrão *join*, implementado como uma instância da classe `JoinPattern`.

O terceiro nível é formado pelo conjunto de filas de invocações de fragmentos. Cada fragmento na classe dá origem a uma fila de invocações daquele fragmento, implementada como instâncias da classe `JoinFragment`.

A estrutura básica do módulo de casamento de padrões pode ser visto na Figura 4.3.

O `Chords` dispõe do conjunto de `JoinFragment` e o conjunto de `JoinPattern` da classe. `JoinPattern` dispõe do conjunto de `JoinFragment` necessários a sua execução. Por sua vez, cada `JoinFragment` dispõe do conjunto de todos os `JoinPattern` que ele pode compor. Assim, o algoritmo básico para o casamento de padrões pode ser visto em Figura 4.4

Toda vez que um fragmento de acorde é chamado, o gerenciador de acordes busca a fila de invocações de fragmentos específica daquele fragmento e enfileira os parâmetros da chamada. Então, para cada padrão *join* em que o fragmento chamado pode participar, é verificado se o padrão está completo, ou seja, se há pelo menos um elemento nas

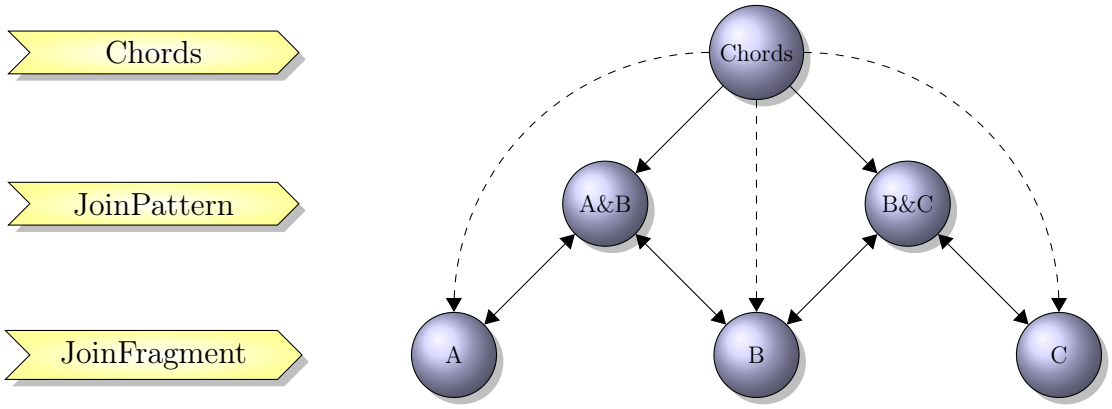

Figura 4.3. Casamento de padrões

```
Input: Invocação de fragmento
início
    obtém(JoinFragment em Chords);
    enfileira(Invocação em JoinFragment);
    para cada JoinPattern de JoinFragment faça
        se completo(JoinPattern) então
            para cada JoinFragment de JoinPattern faça
                desenfileira(JoinFragment);
            fim
            executa(JoinPattern);
            retorna;
        fim
    fim
fim
```

Figura 4.4. Algoritmo de casamento de padrões

filas de todos os seus fragmentos. Se houver, os parâmetros de cada um dos fragmentos são desenfileirados de suas respectivas filas e passados ao padrão *join* que irá usá-los para executar o método acorde correspondente à aquele padrão.

Utilizando a Figura 4.3 e a Figura 4.4, considere a situação inicialmente de todas as filas vazias, então:

1. Ocorre uma chamada ao fragmento A. O gerenciador de acordes localiza a fila A e adiciona essa chamada. A fila A só faz parte do padrão $A\&B$, portanto somente esse padrão é verificado. Esse padrão depende do fragmento B, e a fila B está vazia, o processo termina.

2. Ocorre uma chamada ao fragmento C. O gerenciador de acordes localiza a fila C e adiciona essa chamada. A fila C só faz parte do padrão $B\&C$, portanto

somente esse padrão é verificado. Como esse padrão depende do fragmento B, e a fila B está vazia, o processo termina.

3. Ocorre a chamada ao fragmento B. O gerenciador de acordes localiza a fila B e adiciona essa chamada. A fila B faz parte dos padrões $A\&B$ e $B\&C$, portanto ambos os padrões são verificados. Ambos os padrões estão completos contudo somente um deles será executado. A implementação atual usa a ordem de declaração dos acordes como método de desempate, contudo sistemas baseados em JChords não devem pressupor esse critério. Supondo a execução do padrão $A\&B$, então são extraídos elementos das filas A e B, ficando ambas vazias, e o método acorde relacionado a esse padrão será executado.

4.5 Conclusão

A implementação de JChords utiliza diversas camadas de atuação que se complementam para integrar acordes em uma plataforma Java padrão. Cada camada atua em um momento do ciclo de vida do sistema Java e disponibiliza recursos específicos para as camadas seguintes.

Anotações em código e instrumentação de *bytecodes* fornecem uma técnica poderosa que permite à biblioteca adaptar a sintaxe e a semântica da linguagem para implementar novas funcionalidades ao Java sem comprometer o ambiente padrão de desenvolvimento e execução. Essa técnica possibilita manipular um sistema por meio do próprio sistema sem alterar significativamente o desenvolvimento normal deste.

A camada de desenvolvimento disponibiliza anotações por meio das quais o desenvolvedor descreve os acordes e estruturas de apoio junto ao código fonte Java convencional.

A camada de transformação aplica uma sequência encadeada de filtros ou agentes de transformação que atuam diretamente sobre os *bytecodes*, e utilizam as anotações disponibilizadas pela camada de desenvolvimento e instrumentam as classes com o suporte a acordes.

Por fim, a camada de execução provê a infraestrutura para suportar as classes instrumentadas produzidas na etapa anterior, incluindo o mecanismo de casamento de padrões que coordena a formação de *joins* e a execução dos acordes.

O esforço conjunto de todas as camadas de atuação proporciona uma implementação de acordes por meio de uma biblioteca externa mantendo uma sintaxe apropriada para a estrutura de acordes e com a estrutura sintática de um código Java convencional.

Capítulo 5

Avaliação

Dentre as principais soluções para o uso de acordes nas principais linguagens imperativas orientadas a objetos, JChords ocupa um nicho que anteriormente não era suportado por nenhuma outra solução, ou seja, uma solução baseada em biblioteca para a plataforma Java. Considerando as duas principais linguagens de programação modernas orientadas a objetos, Java e C#, e ainda as duas principais estratégias para implementar acordes nessas linguagens, via biblioteca externa ou via compilador customizado, observa-se a seguinte conformação:

	Biblioteca	Compilador
Java	**JChords**	Join Java
C#	Joins	Cω

Tabela 5.1. Soluções em acordes

Do ponto de vista funcional, excetuando-se os aspectos relacionadas a plataforma e implementação, todas as soluções apresentadas desempenham exatamente o mesmo papel e todas elas desempenham esse papel disponibilizando essencialmente os mesmos elementos básicos. Ou seja, todas elas disponibilizam os recursos necessários a programação baseada em acordes. Os meios usados por cada solução para disponibilizar esses recursos varia de acordo com a plataforma, com a estratégia de implementação adotada e com o desenho da solução. Contudo todas as soluções disponibilizam meios para a criação de fragmentos síncronos e assíncronos e para a especificação de padrões *join* e acordes, além de prover a infraestrutura necessária para gerenciar e executar acordes.

Cada uma das soluções analisadas serve a um cenário de uso específico e atende aos requisitos desse cenário. Dessa forma não é objetivo dessa análise demonstrar a superioridade absoluta de uma solução em relação a outra mas sim demonstrar os

diferenciais significativos de desenho, implementação e plataforma de cada uma das soluções. O objetivo dessa análise é estabelecer critérios para seleção de uma soluções, baseada nos requisitos de um cenário de uso específico. Além disso essa análise objetiva mostrar os cenários e situações em que JChords se apresenta como a solução mais adequada.

Tendo esses objetivos em vista foram selecionados os seguintes critérios de análise:

- Funcionamento em ambiente de desenvolvimento padrão
- Funcionamento em ambiente de execução padrão
- Permite acordes compostos
- Priorização de acordes
- Validação em tempo de compilação
- Otimização em tempo de compilação
- Acordes como métodos
- Multiplataforma
- Não requer declaração de fragmentos síncronos
- Não requer declaração de fragmentos assíncronos
- Não requer inicialização dos fragmentos

A análise que se segue não considera o desempenho em tempo de execução das soluções analisadas. Isso se deve principalmente pela impossibilidade de comparar objetivamente o desempenho de sistemas concorrentes em plataformas distintas, visto que quaisquer diferenças podem ser atribuídas tanto a solução quanto a plataforma.

5.1 Critérios

"Funcionamento em ambiente de desenvolvimento padrão" se refere a capacidade da solução em ser usada em um ambiente de desenvolvimento suportado por grandes empresas ou comunidades, normalmente ambientes homologados pelo próprio provedor da plataforma escolhida. Na indústria de software há um volume considerável de tempo, dinheiro e recursos investidos na criação e manutenção de sistemas. O uso de ferramentas de desenvolvimento não padronizadas representa um risco a esse investimento, seja devido as potenciais deficiências ou ineficiências dessas ferramentas ou pelo risco

de interrupção súbita no suporte e manutenção das mesmas. Dessa forma, o uso de ambientes não padronizados tende a ser rejeitado em um ambiente de produção e por consequência, ferramentas capazes de operar em ambientes de desenvolvimento padrão podem ser as únicas opções nesses casos.

De maneira análoga, "funcionamento em ambiente de execução padrão" se refere a capacidade da solução de ser executada em um ambiente de execução suportado e homologado por grandes empresas ou comunidades. Esse critério se baseia nos mesmos princípios do critério anterior, "funciona em ambiente de desenvolvimento padrão", contudo este critério é ainda mais importante. A adoção de um ambiente de execução customizado que apresente alguma deficiência ou incompatibilidade pode comprometer a estabilidade do servidor de aplicações e por consequência comprometer não só a aplicação que utiliza o ambiente customizado, mas também todas as demais aplicações daquele servidor. Dessa forma, novamente, o uso de ambientes não padronizados tende a não ser aceito em ambientes de produção.

O critério "permite acordes compostos", se refere à capacidade de uma solução em expressar os padrões *join* de maneira composta, ou seja, reusando parte assíncrona do padrão para declarar mais de um acorde. Esse recurso permite agrupar os acordes de modo a melhorar sua organização no código:

```
1 public int get() & positive(int n) {
2   return n;
3 } & negative(int n) {
4   return -n;
5 }
```

Listagem 5.1. Acorde composto

Nesse exemplo temos os padrões `get()& positive(int)` e `get()& negative(int)` sendo ambos descritos por meio de uma única declaração do fragmento síncrono `get()`.

"Priorização de acordes" descreve a capacidade da solução em favorecer a execução de certos acordes quando houver a possibilidade de executar mais de um acorde. O *join-calculus* estabelece que havendo dois padrões *join* possíveis em uma dada expressão um deles deve ser escolhido não-deterministicamente, deixando o critério de escolha a cargo da implementação específica. Nesse critério é considerada a capacidade da solução em permitir declarar qual acorde deve ser executado quando houver a possibilidade de executar mais de um acorde. Vale ressaltar que, como já foi demonstrado na solução do problema do Papai Noel na Subseção 3.3.3, a priorização de acordes sempre pode ser feita por meio da inclusão de acordes auxiliares, sendo que o recurso de priorização de acordes representa apenas uma facilidade ao desenvolvedor.

No critério "validação em tempo de compilação" avalia-se a capacidade da solução

em detectar erros durante a compilação. Todas as soluções apresentadas aprimoram de alguma forma sua linguagem base, e com isso passam a existir novas regras de validação de estrutura e de sintaxe as quais as violações precisam ser identificadas. Esse critério avalia se a solução é capaz de indicar a presença de erros sem a necessidade de executar e testar o sistema. Não está sendo considerado aqui quão efetiva é a detecção de erros da solução, apenas se ela é possível.

O critério "otimização em tempo de compilação" descreve a capacidade da solução em analisar e aprimorar o código criado pelo desenvolvedor de modo a melhorar o desempenho do sistema durante a execução. Vale observar que não é considerado aqui quão efetiva é a otimização, apenas se ela é possível. Além disso está sendo levado em consideração somente a possibilidade de otimizar a execução dos acordes e não da linguagem base como um todo.

"Acordes como métodos" avalia o quão natural é a técnica de integração de acordes com a linguagem base. No paradigma orientado a objetos, toda computação se dá pela troca de mensagens entre objetos. Dessa forma considera-se natural que os acordes sejam implementados como uma abstração da troca de mensagens normal do paradigma orientado a objetos. Soluções que não seguem esse critério tendem a demandar um esforço mental maior de um desenvolvedor devido a necessidade de se adequar o paradigma.

O critério "Multiplataforma" define a capacidade de um sistema funcionar em diversas combinações de *hardware* e sistemas operacionais sem demandar mudanças no código fonte, específicas a cada plataforma. Devido a fatores econômicos e tecnológicos, frequentemente é necessário desenvolver sistemas capazes de executar em uma população heterogênea de dispositivos computacionais. Esse critério avalia a capacidade da solução de viabilizar seu uso em ambientes computacionais heterogêneos sem demandar mudanças no código fonte do sistema.

"Não requer declaração de fragmentos síncronos" e "não requer declaração de fragmentos assíncronos" descrevem a capacidade de uma solução em identificar, por meio das assinaturas dos acordes, os fragmentos síncronos e assíncronos necessários, sem que haja a necessidade de declará-los explicitamente. Em geral, soluções que requerem menos declarações explícitas tendem a ser mais simples de usar pois demandam menos esforço do desenvolvedor e poluem menos o código fonte.

Por fim, o critério "não requer inicialização dos fragmentos" identifica soluções onde não é necessário executar nenhum mecanismo ou rotina de inicialização explícita para habilitar o uso de acordes. Esse critério identifica soluções para as quais além da declaração vista nos critérios anteriores, ainda requerem algum tipo de inicialização dos fragmentos. Novamente soluções que requerem menos inicializações explícitas tendem

a permitir códigos mais simples e menos poluídos.

5.2 JChords

Como já foi visto anteriormente, JChords implementa acordes em Java por meio de uma biblioteca externa. JChords foi desenvolvida para ser usada em um ambiente Java padrão tanto em tempo de desenvolvimento quanto em tempo de execução. São usadas anotações para incorporar a classes e métodos os elementos necessários ao desenvolvimento com acordes e para declarar os padrões *join*.

Sendo uma biblioteca externa, desenvolvida sobre a própria plataforma Java, JChords poderá, em teoria, ser usada com qualquer versão futura do ferramental Java além de se beneficiar de melhorias e aprimoramentos que vierem a ser disponibilizados na plataforma. Além disso, por ser possível utilizá-la com um ambiente de desenvolvimento padrão, JChords pode ser prontamente utilizada em ambientes de produção.

A sintaxe de JChords já foi examinada no Capítulo 3. A Listagem 3.20 exibe, por conveniência, a classe `Group` usada na solução do problema do Papai Noel:

```
1  package usecases.santaclaus.jchords;
2
3  import jchords.annotations.*;
4
5  @Chorded public class Group {
6    @Sync public void entry() {}
7    @Async public void tokens(int n) {}
8    @Sync public void waitt() {}
9    @Async public void allgone() {}
10
11   public void accept(int n) {
12     tokens(n);
13     waitt();
14   }
15
16   @Join({"void entry()", "void tokens(int)"})
17   public void entry_tokens(int n) {
18     if (--n == 0) {
19       allgone();
20     } else {
21       tokens(n);
22     }
```

```
23   }
24
25   @Join({"void waitt()", "void allgone()"}) public void wait_allgone() {}
26 }
```

Listagem 5.2. usecases/santaclaus/jchords/Group.java

Classe `Group`

Em JChords, fragmentos síncronos e assíncronos precisam ser declarados explicitamente. Isso é necessário de modo a garantir que a invocação de um fragmento seja uma invocação de método válida no código. Embora isso possa aumentar o volume de código necessário, essa característica possibilita a adição de trechos auxiliares de código nos fragmentos. Apesar de normalmente esses códigos auxiliares não serem necessários, eles podem ser úteis para controle e monitoramento dos acordes. Além disso, as declarações dos fragmentos podem ser usadas como *breakpoints* nos depuradores disponíveis no ecossistema Java.

Os padrões *join* são declarados em JChords pela anotação `@Join`. Essa anotação recebe como parâmetro um arranjo de *strings*, sendo que cada elemento desse arranjo descreve um dos fragmentos que compõem o padrão. O uso de *strings* para descrever um fragmento possibilita a introdução de erros de digitação que não são detectados em tempo de compilação. Além disso, ferramentas de refatoração, via de regra, não são capazes de ajustar os descritores de fragmentos adequadamente, requerendo a intervenção manual do desenvolvedor.

Por fim, sendo Java uma ferramenta multiplataforma, ou seja, capaz de executar em ambientes computacionais diversos, ferramentas baseadas em Java herdam essa habilidade, o que amplia o universo potencial de aplicações para JChords.

5.3 Joins

A biblioteca Joins proposta por Russo [2006], assim como JChords, implementa acordes por meio de uma biblioteca externa. Diferentemente de JChords, Joins implementa acordes para C# na plataforma .NET padrão.

Joins provê uma interface onde os elementos necessários ao desenvolvimento com acordes são objetos. Ou seja, os fragmentos, padrões *join* e acordes são representados por instâncias de classes específicas, configurados em tempo de execução para desempenhar os papéis das construções para acordes.

O trecho de código da Listagem 5.3 foi adaptado do conjunto de exemplos da distribuição de Joins. A classe apresentada equivale a classe `Group`, chamada aqui de `nway`:

```
1  public class nway {
2    private readonly Asynchronous.Channel<int> tokens;
3    public readonly Synchronous.Channel entry;
4    private readonly Asynchronous.Channel allgone;
5    private readonly Synchronous.Channel wait;
6
7    public void acceptn(int n) {
8      tokens(n);
9      wait();
10   }
11
12   public nway(){
13     Join j = Join.Create();
14
15     j.Initialize(out tokens);
16     j.Initialize(out entry);
17     j.Initialize(out allgone);
18     j.Initialize(out wait);
19
20     j.When(entry).And(tokens).Do(delegate(int n){
21       if (n==1) {
22         allgone();
23       } else {
24         tokens(n-1);
25       }
26     });
27
28       j.When(wait).And(allgone).Do(delegate(){});
29   }
30 }
```

Listagem 5.3. Classe nway

Joins requer que os fragmentos síncronos e assíncronos sejam declarados. É necessário também inicializar esses fragmentos antes que eles possam ser usados, uma vez que se tratam de objetos.

Além disso, os fragmentos só podem receber no máximo um parâmetro. Isso se deve à impossibilidade de tratar classes genéricas com número variável de tipos. Contudo isso pode ser contornado pela utilização de tuplas ou arranjos.

Vale observar que a biblioteca utiliza amplamente os *delegates* disponíveis em C#

para simplificar diversos pontos da sintaxe. Esse não é um recurso que existe em Java.

Os padrões *join* são declarados invocando métodos encadeados na classe *Join*. Essa sintaxe, associada à sintaxe para declarar e inicializar fragmentos, pode ser confusa e diluir a solução do problema. Contudo, pela utilização de objetos e classes genéricas, Joins garante a validade dos nomes e tipos dos fragmentos em tempo de compilação.

Como pode ser constatado, sendo uma implementação por biblioteca externa Joins compartilha muitas das características de JChords. A principal diferença entre as duas bibliotecas, além da plataforma, é a opção por utilizar anotações ou objectificação, o que resulta na diferença na abordagem sintática das bibliotecas.

5.4 Join Java

Join Java, proposto por Itzstein [2005], é uma das primeira implementações de acordes em linguagens imperativas orientadas a objetos. Assim como JChords, Join Java se baseia na plataforma Java. Ao contrário de JChords, Join Java foi implementado como um compilador customizado.

As principais desvantagens da implementação via compilador customizado é o alto risco de obsolência. A menos que haja um esforço de sincronização efetivo, compiladores customizados tendem a apresentar incompatibilidades e a não incorporar melhorias e aprimoramentos implementados no ferramental de desenvolvimento padrão. Além disso, evoluções da linguagem podem demandar um grande esforço para atualizar o compilador, mesmo não havendo qualquer mudança na estrutura de acordes em si.

Por outro lado, Join Java apresenta a sintaxe mais limpa e direta dentre as soluções analisadas. Uma classe com acorde em Join Java não requer declaração ou inicialização de fragmentos. Os padrões *join* são declarados diretamente nas assinaturas dos acordes e todos os fragmentos necessários são inferidos por meio dessas assinaturas. A Listagem 5.4 mostra a mesma classe `Group` pra o problema do Papai Noel, implementada em Join Java:

```
1  final public class Group {
2    public void accept(int n) {
3      tokens(n);
4      waitt();
5    }
6
7    public void entry() & tokens(int n) {
8      if (--n == 0) {
9        allgone();
10     } else {
```

```
11        tokens(n);
12      }
13    }
14
15    public void waitt() & allgone() {}
16  }
```

Listagem 5.4. Group

São introduzidas as palavras chave `signal` e `ordered`. `signal` serve como tipo de retorno para acordes que não possuem fragmentos síncronos. `ordered` permite declarar classes onde a ordem em que os acordes aparecem determina a prioridade dos mesmos, ou seja, dado dois padrões *join* possíveis, será executado aquele que tiver sido declarado primeiro no corpo da classe.

Sendo uma solução baseada em compilador, toda a verificação e validação da sintaxe é feita em tempo de compilação. Além disso, uma vez que o compilador tem uma visão global do sistema, é possível aplicar otimizações mais agressivas aos *bytecodes* gerados.

Apesar de exigir um ambiente de desenvolvimento customizado, Join Java gera *bytecodes* de acordo com a especificação padrão da máquina virtual Java e, por consequência, classes compiladas com Join Java operam em ambientes de execução padrão.

Sintaticamente, Join Java é uma implementação mais limpa e direta de acordes. Em contrapartida o uso de compiladores customizados pode inviabilizar o uso de Join Java em situações onde JChords pode ser aplicado sem restrições.

5.5 Cω

Cω proposto por Benton et al. [2004] é uma evolução do Polyphonic C# proposto por Benton et al. [2002] e implementa acordes em C# por meio de um compilador customizado. Assim como Join Java, Cω é suceptível as dificuldades em utilizar ferramentas customizadas em ambientes produtivos. Por outro lado possui uma sintaxe mais limpa e direta que implementações por biblioteca, assim como dispõe de melhores recursos para validação e otimização em tempo de compilação.

A Listagem 5.5 ilustra a classe `nway`, equivalente a `Group` na solução do problema do Papai Noel. O trecho de código mostrado foi adaptado dos exemplos que acompanham a distribuição de Cω:

```
1  public class nway {
2    async tokens(int n);
3    async allgone();
4
```

```
5   public void acceptn(int n) {
6     tokens(n);
7     wait();
8   }
9
10  public void entry() & tokens(int n) {
11    if (n==1) {
12      allgone();
13    } else {
14      tokens(n-1);
15    }
16  }
17
18  void wait() & allgone() {}
19 }
```

Listagem 5.5. Classe `nway`

A linguagem base é aprimorada pela inclusão da palavra reservada `async` usada como tipo de retorno para a declaração dos fragmento assíncronos. Os padrões *join* são declarados diretamente na assinatura do acorde, sendo que existe o suporte a acordes compostos.

Ao contrário de Join Java e à semelhança de Joins e JChords, Cω requer a declaração dos fragmentos assíncronos. Os fragmentos síncronos não precisam ser declarados.

Cω incorpora diversos outros recursos além de acordes. Do ponto de vista da implementação de acordes Cω equivale a Join Java em quase todos os aspectos. Consequentemente, excetuando-se a plataforma, Cω se compara a JChords da mesma maneira que Join Java.

5.6 Conclusão

As principais características de cada uma das soluções analisadas é sumarizada na Tabela 5.2.

As soluções baseadas em compiladores customizados se destacam quanto a sintaxe e robustez, uma vez que a sintaxe da linguagem base foi explicitamente modificada para acomodar os acordes.

Por outro lado, as inconveniências associadas ao uso de compiladores customizados podem inviabilizar o uso dessas soluções em ambientes não experimentais. Nesse caso as soluções baseadas em biblioteca demandam um custo adicional na sintaxe em troca

	JChords	Joins	Join Java	Cω
Ambiente de desenvolvimento padrão	✓	✓		
Ambiente de execução padrão	✓	✓	✓	✓
Acordes compostos				✓
Priorização de acordes			✓	✓
Validação em tempo de compilação		✓	✓	✓
Otimização em tempo de compilação			✓	✓
Acordes como métodos	✓		✓	✓
Multiplataforma	✓		✓	
Não declara fragmentos síncronos			✓	✓
Não declara fragmentos assíncronos			✓	
Não inicializa fragmentos	✓		✓	✓

Tabela 5.2. Sumário da avaliação

de serem aptas a funcionarem em ambientes padrão.

JChords se destaca por oferecer uma sintaxe razoavelmente semelhante às soluções baseadas em compiladores customizados, porém implementada como uma biblioteca externa. Além de ocupar o nicho, anteriormente vazio, das bibliotecas para acordes na plataforma JAVA.

Capítulo 6

Considerações Finais

Desenvolver sistemas capazes de operar paralelamente, que possam ser facilmente escalonados entre muitos processadores de forma efetiva é um dos grandes desafios da computação moderna. Contudo, os principais mecanismos usados para expressar paralelismo e concorrência nas principais linguagens de programação se mostram inadequados à tarefa que se apresenta, tendo em vista o alto grau de dificuldade normalmente associado ao desenvolvimento de sistemas concorrentes.

Sistemas concorrentes se mostram, ainda hoje, complexos de projetar e construir, susceptíveis a erros grosseiros mas que se manifestam de maneira sutil e intermitente. Difíceis de serem detectados e corrigidos com o ferramental técnico disponível.

As *threads*, hoje a principal construção para expressar concorrência nas linguagens de programação imperativas, são construções de baixo nível de abstração que não deveriam ser usadas diretamente, mas sim servindo como base para construções de mais alto nível. Isso segue a tendência das principais linguagens modernas, que buscam poupar o desenvolvedor do encargo de micro-gerenciar os recursos do sistema.

Os acordes oferecem um mecanismo de alto nível de abstração para expressar sistemas concorrentes. Eles integram com sucesso a solidez conceitual da álgebra de processo *join-calculus* ao paradigma imperativo orientado a objetos, oferecendo uma forma elegante, prática e segura de expressar soluções para problemas de concorrência e paralelismo.

Diversas soluções tem sido propostas para implementar acordes nas principais linguagens de programação, em especial C# e Java. A implementação de acordes geralmente é feita pela criação de compiladores customizados ou pela criação de bibliotecas externas. A implementação por meio de compiladores customizados, em tese, é mais adequada a técnicas maduras e estáveis, onde não se esperam grandes desenvolvimentos ou evoluções. Por outro lado, a implementação por biblioteca é mais adequada a técnicas novas e em evolução.

Dessa forma, nesse trabalho mostrou-se o projeto, a implementação e aplicação de JChords, uma iniciativa para prover acordes em Java por meio de uma biblioteca externa. Almejou-se com esse esforço preencher o espaço atualmente vazio de acordes via biblioteca em Java, e integrar os acordes ao ecossistema e ao ferramental Java de maneira gradual e flexível. Encorajando a experimentação e evolução.

A biblioteca JChords permite expressar as soluções para problemas de concorrência em um nível mais alto de abstração, de modo mais simples, direto e seguro. Liberando o desenvolvedor do gerenciamento de *threads* e permitindo que ele se concentre nas demandas do problema em si. Apesar das limitações que uma implementação por biblioteca impôs, JChords apresentou resultados bastante satisfatórios em termos de estrutura e expressividade. Mostrando-se adequado para substituir as *threads* na maioria das situações estudadas.

A tarefa de implementar acordes em Java via biblioteca demandou mecanismos que exploram os novos recursos do Java 5 que viabilizaram a descrição e implementação de soluções baseadas em acordes sem demandar alterações na linguagem.

A implementação de JChords foi realizada de forma modular, utilizando três camadas de atuação que se complementam para integrar o resultado final. A camada de desenvolvimento baseada em anotações, a camada de transformação baseada na instrumentação e manipulação das classes compiladas, e a camada de execução baseada em reflexão. A atuação em conjunto dessas camadas proporciona a anexação da sintaxes e semânticas necessárias para o desenvolvimento de sistemas baseados em acordes sem demandar alterações na plataforma Java padrão.

6.1 Trabalhos Futuros

O presente trabalho é mais um passo para um suporte efetivo à programação baseada em acordes na plataforma Java. Há ainda uma grande variedade de oportunidades de trabalhos tanto internos quanto externos à JChords.

Nesse primeiro momento, JChords é principalmente uma prova de conceito de uma implementação de acordes baseada em biblioteca. O principal foco até o momento foi a usabilidade das construções que suportam acordes e a arquitetura para viabilizar as intervenções necessárias. Assim, muitos aspectos da implementação podem ser otimizados e aperfeiçoados.

6.1.1 Otimização

Um ponto de otimização bastante relevante é o mecanismo de casamento de padrões. Como já foi visto, JChords implementa a mesma estratégia de casamento de padrões

proposta por Itzstein [2005]. Por ser central ao funcionamento da biblioteca, melhorias nesse mecanismo seriam bastante benéficas ao sistema como um todo.

Outro ponto significativo diz respeito a mecânica de invocação de métodos assíncronos. Os métodos assíncronos representam uma grande parcela das mensagens transportadas em JChords. É possível que haja ganhos de desempenho significativos pela substituição da mecânica de invocação por reflexão em favor do acesso direto via manipulação de *bytecode*.

6.1.2 Validação

Um outro aspecto de JChords que pode ser aprimorado é a validação das regras de acordes em tempo de compilação. Como foi visto, devido a maneira como a biblioteca foi concebida, erros na composição dos padrões *join* ou outras violações das regras da biblioteca só são detectados em tempo de carga. O uso de processadores de anotações pode fornecer os meios para detectar esses erros em tempo de compilação.

A versão 5 do Java disponibiliza, junto ao suporte a anotações, uma ferramenta que permite anexar processadores de anotações ao compilador Java. Esses processadores não possibilitam manipular ou alterar as classes geradas, consequentemente eles não são adequados como mecanismos de transformação de JChords. Contudo os processadores de anotações podem inspecionar as classes em tempo de compilação e com isso podem ser usados para verificar e validar os acordes em tempo de compilação, emitindo avisos de erros se necessário.

6.1.3 Herança e Polimorfismo

Um outro aspecto significativo para a integração de acordes em linguagens orientadas a objetos é o suporte a herança. Na atual encarnação de JChords não são feitas provisões para o tratamento de hierarquias de classes com acordes. A herança entre classes com acordes é permitida, contudo não há suporte para herança de fragmentos ou padrões *join*, nem suporte à sobrescrita ou polimorfismo de fragmentos ou acordes. Ou seja a herança é suportada desde que os elementos baseados em acordes não sejam compartilhados entre as classes.

Itzstein [2005] optou por não permitir a herança com classes com acordes ao passo que Benton et al. [2004] optou por permitir herança e tratar as anomalias resultantes da integração da herança e acordes [Matsuoka & Yonezawa, 1993]. A experimentação com o suporte a hierarquias permitiria uma análise mais refinada dos meios e métodos para implementá-lo efetivamente em JChords.

6.1.4 Padrões *Join*

Sob muitos aspectos, podemos considerar os padrões *join* como expressões lógicas conjuntivas. O padrão `A()& B()` requer que exista uma invocação do fragmento `A()` e uma invocação do fragmento `B()` para permitir a execução do acorde.

Usando apenas adições sintáticas à biblioteca, talvez seja possível permitir padrões disjuntivos do tipo `A()| B()`, ou seja, um padrão que requer o fragmento `A()` ou o fragmento `B()`. Ou ainda podemos compor padrões por negação do tipo `A()& ~B()` para possibilitar a execução de um acorde somente na ausência de um determinado fragmento.

Essas adições podem simplificar certos padrões que de outra maneira exigiriam dois ou mais acordes para serem descritos.

6.1.5 Padrões de Sincronização

Durante o desenvolvimento dos casos de uso de acordes, foram percebidos alguns padrões de sincronização recorrentes. O estudo e catalogação desses padrões devem permitir estabelecer o repertório de melhores práticas no desenvolvimento de sistemas baseados em acordes.

6.1.6 Performance

Mensurar o desempenho de sistemas concorrentes é uma tarefa complexa. A medição direta de *throughput* em um equipamento mono-processado não é efetiva uma vez que devem ser consideradas as relações entre a capacidade de processamento do *hardware*, a qualidade da implementação do *software* e a capacidade do sistema em escalonar-se graciosamente no maior número possível de unidades de processamento.

Em teoria, um sistema perfeito desenvolvido utilizando *threads* será sempre mais rápido que um outro sistema igualmente perfeito desenvolvido usando acordes, uma vez que o sistema baseado em acordes dispende esforço na gestão de fluxos de execução.

Contudo, na prática, é difícil desenvolver sistemas concorrentes de larga escala utilizando *threads* que exibam uma implementação perfeita. Conjectura-se que sistemas baseados em acordes sejam mais simples de desenvolver e implementam uma gestão mais efetiva de fluxos paralelos de execução. Por consequência, supõe-se que sistemas baseados em acordes demandem menor tempo de desenvolvimento, uma vez superada a curva de aprendizado, e exibam maior paralelismo que os sistemas baseados em *threads*.

Dessa forma, a realização de testes de campo comparativos empregando grupos de desenvolvedores utilizando *threads* e acordes na solução de problemas práticos deve

prover uma medida do real impacto da aplicação de acordes no desempenho de sistemas paralelos e concorrentes.

6.2 Conclusão

Considera-se que esse trabalho cumpriu seus objetivos demonstrando a necessidade e a viabilidade de construções de alto nível para programação concorrente em Java mantendo a arquitetura padrão da plataforma. A programação baseada em acordes é uma técnica relativamente recente porém mostra grande potencial para ser mais um passo em direção a um arcabouço efetivo para o desenvolvimento de sistemas paralelos.

Apêndice A

Códigos Fonte

A.1 Biblioteca

```
1  package jchords;
2
3  import java.lang.reflect.Method;
4
5  public class AsyncCall {
6    private class Call extends Thread {
7      private final Object object;
8      private final Object[] args;
9
10     Call(Object object, Object... args) {
11       this.object = object;
12       this.args = args;
13     }
14
15     public void  run() {
16       try {
17
18         method.invoke(object, args);
19       } catch (Exception ex) {
20         throw new RuntimeException(ex);
21       }
22     }
23   }
24
25   private final Method method;
```

```
26
27   public AsyncCall(Class<?> cls, String name, Class<?>... args) {
28     try {
29       method = cls.getDeclaredMethod(name, args);
30       method.setAccessible(true);
31     } catch (NoSuchMethodException ex) {
32       throw new RuntimeException(ex);
33     }
34   }
35
36   public void invoke(Object target, Object... args) {
37     new Call(target, args).start();
38   }
39
40   public static void invoke(
41     Class<?> clazz, String name, Class<?>[] args, Object object, Object[]
         values
42   ) {
43     new AsyncCall(clazz, name, args).invoke(object, values);
44   }
45 }
```

Listagem A.1. jchords/AsyncCall.java

```
1  package jchords;
2
3  import org.objectweb.asm.commons.Method;
4
5  public class Call {
6    Method method;
7    Object[] args;
8    Object result = null;
9    boolean ready = false;
10
11   public Call(Method method, Object[] args) {
12     this.method = method;
13     this.args = args;
14   }
15 }
```

Listagem A.2. jchords/Call.java

```
1  package jchords;
2
3  import static jchords.asm.MethodUtils.getMethod;
4  import java.util.*;
5  import jchords.exceptions.UnpreparedChordedClass;
6  import org.objectweb.asm.commons.Method;
7
8  public class Chords {
9    private final Object object;
10   private Map<Method, JoinFragment> fragments =
11     new HashMap<Method, JoinFragment>();
12
13   public Chords(Object object, JoinDescriptor... joins) {
14     this.object = object;
15     for (JoinDescriptor join : joins) {
16       JoinPattern pattern = providePattern(join.method);
17       for (Method method : join.fragments) {
18         JoinFragment fragment = provideFragment(method);
19         fragment.patterns.add(pattern);
20         pattern.fragments.add(fragment);
21       }
22     }
23   }
24
25   private JoinPattern providePattern(Method method) {
26     try {
27       return new JoinPattern(object, getMethod(object.getClass(), method));
28     } catch(Exception ex) {
29       throw new RuntimeException(ex);
30     }
31   }
32
33   private JoinFragment provideFragment(Method method) {
34     JoinFragment fragment = fragments.get(method);
35     if(fragment == null) {
36       fragment = new JoinFragment(method);
37       fragments.put(method, fragment);
38     }
39     return fragment;
```

```
40    }
41
42    private Call add(String desc, Object... args) {
43      return add(Method.getMethod(desc), args);
44    }
45
46    private Call add(Method method, Object... args) {
47      try {
48        return fragments.containsKey(method) ?
49          fragments.get(method).add(args) : null;
50      } finally {
51        notifyAll();
52      }
53    }
54
55    public synchronized void addAsync(String desc, Object... args) {
56      add(desc, args);
57    }
58
59    public synchronized Object addSync(String desc, Object... args) {
60      Call call = add(desc, args);
61      while (call.ready != true) {
62        try {
63          wait();
64        } catch (java.lang.InterruptedException ex) {
65          throw new RuntimeException(ex);
66        }
67      }
68      return call.result;
69    }
70
71    public static <T> T result(Class<T> cls) {
72      throw new UnpreparedChordedClass();
73    }
74  }
```

Listagem A.3. jchords/Chords.java

```
1  package jchords;
2
3  import java.util.Collection;
```

```
4  import org.objectweb.asm.commons.Method;
5
6  public class JoinDescriptor {
7    public final Method method;
8    public final Method[] fragments;
9
10   public JoinDescriptor(Method method, Method... fragments) {
11     this.method = method;
12     this.fragments = fragments;
13   }
14
15   public JoinDescriptor(Method method, Collection<Method> fragments) {
16     this(method, fragments.toArray(new Method[fragments.size()]));
17   }
18
19   public JoinDescriptor(String method, String... fragments) {
20     this.method = Method.getMethod(method);
21
22     this.fragments = new Method[fragments.length];
23     for(int index = 0; index < fragments.length; ++index)
24     {
25       this.fragments[index] = Method.getMethod(fragments[index]);
26     }
27   }
28
29 }
```

Listagem A.4. jchords/JoinDescriptor.java

```
1  package jchords;
2  import java.util.*;
3  import org.objectweb.asm.commons.Method;
4
5  public class JoinFragment {
6    public final Method method;
7    public final Deque<Call> calls = new LinkedList<Call>();
8    public final List<JoinPattern> patterns = new ArrayList<JoinPattern>();
9
10   public JoinFragment(Method method) {
11     this.method = method;
12   }
```

```
13
14    public Call add(Object[] args) {
15      Call call = new Call(method, args);
16      calls.addLast(call);
17      broadcast();
18      return call;
19    }
20
21    public void broadcast() {
22      for(JoinPattern pattern : patterns) {
23        pattern.brodcast();
24      }
25    }
26
27    public boolean isReady(int times) {
28      return calls.size() >= times;
29    }
30
31    public Call pop() {
32      return calls.removeFirst();
33    }
34  }
```

Listagem A.5. jchords/JoinFragment.java

```
1  package jchords;
2
3  import java.lang.reflect.Method;
4  import java.util.*;
5  import jchords.util.Utils;
6
7  public class JoinPattern {
8    public final Object object;
9    public final Method method;
10   public final List<JoinFragment> fragments = new ArrayList<JoinFragment>();
11
12   public JoinPattern(Object object, Method method) {
13     this.object = object;
14     this.method = method;
15   }
16
```

```
17   public void brodcast() {
18     if(isReady()) { execute(); }
19   }
20
21   private void execute() {
22     List<Call> calls = popFragments();
23     broadcast(calls, invoke(getArguments(calls)));
24   }
25
26   private void broadcast(List<Call> calls, Object result) {
27     for(Call call : calls) {
28       call.result = result;
29       call.ready = true;
30     }
31   }
32
33   private List<Object> getArguments(List<Call> calls) {
34     List<Object> args = new LinkedList<Object>();
35     for(Call call : calls)
36     {
37       Collections.addAll(args, call.args);
38     }
39     return args;
40   }
41
42   private List<Call> popFragments() {
43     List<Call> calls = new LinkedList<Call>();
44     for(JoinFragment fragment : fragments)
45     {
46       calls.add(fragment.pop());
47     }
48     return calls;
49   }
50
51   private Object invoke(List<Object> args) {
52     try {
53       return method.invoke(object, args.toArray(new Object[args.size()]));
54     } catch(Exception ex) {
55       throw new RuntimeException(ex);
56     }
```

```
57   }
58
59   private boolean isReady() {
60     Map<JoinFragment, Integer> counts = new HashMap<JoinFragment, Integer>();
61     for(JoinFragment fragment : fragments) {
62       int count = Utils.safe(counts.get(fragment), 0) + 1;
63       counts.put(fragment, count);
64       if(!fragment.isReady(count)) {
65         return false;
66       }
67     }
68     return true;
69   }
70
71 }
```

Listagem A.6. jchords/JoinPattern.java

```
1  package jchords.agents;
2
3  import static jchords.util.Utils.cast;
4  import java.util.*;
5  import jchords.annotations.Async;
6  import jchords.asm.ClassEnhancer;
7  import jchords.util.Constants;
8  import org.objectweb.asm.*;
9  import org.objectweb.asm.commons.*;
10 import org.objectweb.asm.tree.*;
11
12 public class AsyncTransformer extends ClassEnhancer implements Constants {
13
14   private class MethodTransformer extends MethodNode {
15     private static final String ASYNC_PREFIX = "Async$";
16
17     public MethodTransformer(int access, String name, String desc,
18       String signature, String[] exceptions) {
19       super(access, name, desc, signature, exceptions);
20     }
21
22     public void createAsyncMethod() {
23       GeneratorAdapter mg = new GeneratorAdapter(
```

```
24            access, getMethod(), signature, getExceptions(), cv
25        );
26        mg.visitCode();
27        mg.push(type);
28        mg.push(name = ASYNC_PREFIX + name);
29        loadArgTypeArray(mg);
30        mg.loadThis();
31        mg.loadArgArray();
32        mg.invokeStatic(ASYNCCALL_TYPE, ASYNCCALL_STATIC_INVOKE);
33        mg.returnValue();
34        mg.endMethod();
35      }
36
37      private void loadArgTypeArray(GeneratorAdapter mg) {
38        Type[] args = Type.getArgumentTypes(desc);
39        mg.push(args.length);
40        mg.newArray(CLASS_TYPE);
41        for (int index = 0; index < args.length; ++index) {
42          mg.dup();
43          mg.push(index);
44          mg.push(args[index]);
45          mg.arrayStore(CLASS_TYPE);
46        }
47      }
48
49      private Method getMethod() {
50        return new Method(name, desc);
51      }
52
53      private Type[] getExceptions() {
54        List<Type> list = new LinkedList<Type>();
55        for (Object type : exceptions) {
56          list.add(Type.getObjectType(type.toString()));
57        }
58        return list.toArray(new Type[list.size()]);
59      }
60
61      AnnotationNode getAnnotation(Class<?> clazz) {
62        for (AnnotationNode annotation :
63          cast(AnnotationNode.class, this.visibleAnnotations)
```

```
64        ) {
65          if(annotation.desc.equals(Type.getDescriptor(clazz))) {
66            return annotation;
67          }
68        }
69        return null;
70      }
71
72      @Override
73      public void visitEnd() {
74        if(getAnnotation(Async.class) != null) { createAsyncMethod(); }
75        accept(cv);
76      }
77    }
78
79    public AsyncTransformer(ClassVisitor visitor) {
80      super(visitor);
81    }
82
83    @Override public MethodVisitor visitMethod(
84      int access, String name, String desc, String sig, String[] exceptions
85    ) {
86      return new MethodTransformer(access, name, desc, sig, exceptions);
87    }
88  }
```

Listagem A.7. jchords/agents/AsyncTransformer.java

```
1  package jchords.agents;
2
3  import java.util.*;
4  import jchords.JoinDescriptor;
5  import jchords.asm.*;
6  import jchords.util.Constants;
7  import org.objectweb.asm.*;
8  import org.objectweb.asm.commons.*;
9
10 public class JoinTransformer extends ClassEnhancer implements Constants {
11   protected List<JoinDescriptor> joins = new LinkedList<JoinDescriptor>();
12
13   class ManagerMethod extends MethodEnhacer {
```

```
14
15      protected ManagerMethod(
16        MethodVisitor mv, int access, String name, String desc
17      ) {
18        super(mv, access, name, desc);
19        active = isConstructor(method);
20      }
21
22      @Override protected void onMethodEnter() {
23        if(active) {
24          loadThis();
25          invokeVirtual(type, CHORDS_INIT);
26          super.onMethodEnter();
27        }
28      }
29    }
30
31    class SyncMethod extends MethodEnhacer {
32      private boolean enhanced = false;
33
34      protected SyncMethod(MethodVisitor mv, int access, String name, String
            desc) {
35        super(mv, access, name, desc);
36      }
37
38      @Override public AnnotationVisitor visitAnnotation(String desc, boolean
            visible) {
39        if(desc.equals(SYNC_TYPE.getDescriptor()))
40        {
41          active = true;
42        }
43        return super.visitAnnotation(desc, visible);
44      }
45
46      @Override public void visitMethodInsn(
47        int opcode, String owner, String name, String desc) {
48        if(
49          active &&
50          opcode == INVOKESTATIC &&
51          owner.equals(CHORDS_TYPE.getInternalName()) &&
```

```
52          name.equals(CHORDS_RESULT.getName()) &&
53          desc.equals(CHORDS_RESULT.getDescriptor())
54        ) {
55          pop();
56          dispatch();
57          enhanced = true;
58        }
59        else
60        {
61          super.visitMethodInsn(opcode, owner, name, desc);
62        }
63      }
64
65      @Override protected void onMethodExit(int opcode) {
66        if(active && opcode != ATHROW && !enhanced) {
67          dispatch();
68          pop();
69        }
70        super.onMethodExit(opcode);
71      }
72
73      private void dispatch() {
74        loadThis();
75        getField(type, MANAGER_FIELD_NAME, CHORDS_TYPE);
76        push(getDecl(method));
77        loadArgArray();
78        invokeVirtual(CHORDS_TYPE, CHORDS_ADD_SYNC);
79      }
80    };
81
82    class AsyncMethod extends MethodEnhacer {
83      protected AsyncMethod(MethodVisitor mv, int access, String name, String
          desc) {
84        super(mv, access, name, desc);
85      }
86
87      @Override public AnnotationVisitor visitAnnotation(String desc, boolean
          visible) {
88        if(desc.equals(ASYNC_TYPE.getDescriptor())) {
89          active = true;
```

```
90        }
91        return super.visitAnnotation(desc, visible);
92      }
93
94      @Override protected void onMethodExit(int opcode) {
95
96        if(active && opcode != ATHROW) {
97          loadThis();
98          getField(type, MANAGER_FIELD_NAME, CHORDS_TYPE);
99          push(getDecl(method));
100         loadArgArray();
101         invokeVirtual(CHORDS_TYPE, CHORDS_ADD_ASYNC);
102       }
103       super.onMethodExit(opcode);
104     }
105   };
106
107   class JoinMethod extends MethodEnhacer {
108     class JoinAnnotation extends AnnotationAdapter {
109       List<Method> fragments = new LinkedList<Method>();
110
111       public JoinAnnotation(AnnotationVisitor av) { super(av); }
112
113       public AnnotationVisitor visitArray(String name) {
114         AnnotationVisitor av = super.visitArray(name);
115         if(name.equals("value"))
116         {
117           av = new AnnotationAdapter(av) {
118             public void visit(String name, Object value) {
119               fragments.add(Method.getMethod(value.toString()));
120               super.visit(name, value);
121             }
122           };
123         }
124         return av;
125       }
126
127       public void visitEnd() {
128         joins.add(new JoinDescriptor(method, fragments));
129         super.visitEnd();
```

```
130       }
131     };
132
133     public JoinMethod(MethodVisitor mv, int access, String name, String desc)
            {
134       super(mv, access, name, desc);
135     }
136
137     public AnnotationVisitor visitAnnotation(String desc, boolean visible) {
138       AnnotationVisitor av = super.visitAnnotation(desc, visible);
139       if(desc.equals(JOIN_TYPE.getDescriptor())) {
140         av = new JoinAnnotation(av);
141       }
142       return av;
143     }
144   };
145
146   public JoinTransformer(ClassVisitor cv) { super(cv); }
147
148   public AnnotationVisitor visitAnnotation(String name, boolean visible) {
149     if(name.equals(CHORDED_TYPE.getDescriptor())) {
150       this.active = true;
151     }
152     return super.visitAnnotation(name, visible);
153   }
154
155   public MethodVisitor visitMethod(
156     int access, String name, String desc, String sig, String[] exceptions
157   ) {
158     MethodVisitor mv = super.visitMethod(access, name, desc, sig, exceptions)
          ;
159     if(active) {
160       mv = new ManagerMethod(mv, access, name, desc);
161       mv = new AsyncMethod(mv, access, name, desc);
162       mv = new SyncMethod(mv, access, name, desc);
163       mv = new JoinMethod(mv, access, name, desc);
164     }
165     return mv;
166   }
167
```

```
168   public void visitEnd() {
169     if(active) { createManager(); }
170     super.visitEnd();
171   }
172
173   private void createManager() {
174     FieldVisitor fv = visitField(ACC_PROTECTED, MANAGER_FIELD_NAME,
          CHORDS_TYPE.getDescriptor(), null, null);
175     fv.visitEnd();
176
177     GeneratorAdapter mg = new GeneratorAdapter(
178       ACC_PROTECTED, CHORDS_INIT, null, null, cv
179     );
180     mg.visitCode();
181     mg.loadThis();
182     mg.newInstance(CHORDS_TYPE);
183     mg.dup();
184     mg.loadThis();
185     mg.push(joins.size());
186     mg.newArray(JOINDESCRIPTOR_TYPE);
187     for(int idx = 0; idx < joins.size(); ++idx) {
188       JoinDescriptor pattern = joins.get(idx);
189       mg.dup();
190       mg.push(idx);
191       mg.newInstance(JOINDESCRIPTOR_TYPE);
192       mg.dup();
193       mg.push(MethodEnhacer.getDecl(pattern.method));
194       mg.push(pattern.fragments.length);
195       mg.newArray(STRING_TYPE);
196       for(int index = 0; index < pattern.fragments.length; ++index) {
197         mg.dup();
198         mg.push(index);
199         mg.push(MethodEnhacer.getDecl(pattern.fragments[index]));
200         mg.arrayStore(STRING_TYPE);
201       }
202       mg.invokeConstructor(JOINDESCRIPTOR_TYPE, JOINDESCRIPTOR_CTOR);
203       mg.arrayStore(JOINDESCRIPTOR_TYPE);
204     }
205     mg.invokeConstructor(CHORDS_TYPE, CHORDS_CTOR);
206     mg.putField(type, MANAGER_FIELD_NAME, CHORDS_TYPE);
```

```
207     mg.returnValue();
208     mg.endMethod();
209   }
210 }
```

Listagem A.8. jchords/agents/JoinTransformer.java

```
1  package jchords.agents;
2
3  import static java.lang.Boolean.parseBoolean;
4  import static java.lang.System.getProperty;
5  import static org.objectweb.asm.ClassReader.*;
6  import static org.objectweb.asm.ClassWriter.COMPUTE_FRAMES;
7  import java.io.PrintWriter;
8  import java.lang.instrument.*;
9  import java.security.ProtectionDomain;
10 import org.objectweb.asm.*;
11 import org.objectweb.asm.util.*;
12
13 public class Transformer implements ClassFileTransformer {
14   public static final boolean SHOW_BYTECODE = parseBoolean(
15     getProperty("jchords.showbytecode")
16   );
17
18   public byte[] transform(ClassLoader loader, String name, Class cls,
19     ProtectionDomain domain, byte[] bytes) {
20     ClassReader reader = new ClassReader(bytes);
21     ClassWriter writer = new ClassWriter(reader, COMPUTE_FRAMES);
22     try {
23       reader.accept(adapt(writer), SKIP_FRAMES);
24     } catch (Throwable ex) {
25       ex.printStackTrace();
26     }
27     return writer.toByteArray();
28   }
29
30   private ClassVisitor adapt(ClassVisitor cv) {
31     if (SHOW_BYTECODE) {
32       cv = new TraceClassVisitor(cv, new PrintWriter(System.out));
33     }
34     cv = new CheckClassAdapter(cv);
```

```
35      cv = new AsyncTransformer(cv);
36      cv = new JoinTransformer(cv);
37      return cv;
38    }
39
40    public static void premain(String args, Instrumentation instrumentation) {
41      instrumentation.addTransformer(new Transformer());
42    }
43  }
```

Listagem A.9. jchords/agents/Transformer.java

```
1  package jchords.annotations;
2
3  import java.lang.annotation.*;
4
5  @Retention(RetentionPolicy.RUNTIME)
6  @Target(ElementType.METHOD)
7  public @interface Async {}
```

Listagem A.10. jchords/annotations/Async.java

```
1  package jchords.annotations;
2
3  import java.lang.annotation.*;
4
5  @Retention(RetentionPolicy.RUNTIME)
6  @Target(ElementType.TYPE)
7  public @interface Chorded {}
```

Listagem A.11. jchords/annotations/Chorded.java

```
1  package jchords.annotations;
2
3  import java.lang.annotation.*;
4
5  @Retention(RetentionPolicy.RUNTIME)
6  @Target(ElementType.METHOD)
7  public @interface Join {
8    String[] value();
9  }
```

Listagem A.12. jchords/annotations/Join.java

```
1  package jchords.annotations;
2
3  import java.lang.annotation.*;
4
5  @Retention(RetentionPolicy.RUNTIME)
6  @Target(ElementType.METHOD)
7  public @interface Sync {}
```

Listagem A.13. jchords/annotations/Sync.java

```
1  package jchords.util;
2
3  import jchords.*;
4  import jchords.annotations.*;
5  import jchords.annotations.Join;
6  import org.objectweb.asm.*;
7  import org.objectweb.asm.commons.Method;
8
9  public interface Constants extends Opcodes {
10   Type STRING_TYPE = Type.getType(String.class);
11   Type CLASS_TYPE = Type.getType(Class.class);
12   Type OBJECT_TYPE = Type.getType(Object.class);
13
14   Type JOIN_TYPE = Type.getType(Join.class);
15   Type SYNC_TYPE = Type.getType(Sync.class);
16   Type ASYNC_TYPE = Type.getType(Async.class);
17   Type ASYNCCALL_TYPE = Type.getType(AsyncCall.class);
18   Type CHORDED_TYPE = Type.getType(Chorded.class);
19   Type CHORDS_TYPE = Type.getType(Chords.class);
20   Type JOINPATTERN_TYPE = Type.getType(JoinPattern.class);
21   Type JOINDESCRIPTOR_TYPE = Type.getType(JoinDescriptor.class);
22
23   Method ASYNCCALL_STATIC_INVOKE = Method.getMethod(
24     "void invoke(Class, String, Class[], Object, Object[])"
25   );
26   Method JOINDESCRIPTOR_CTOR = Method.getMethod(
27     "void <init>(String, String[])"
28   );
29   Method CHORDS_CTOR = Method.getMethod(
30     "void <init>(Object,jchords.JoinDescriptor[])"
```

```
31   );
32   Method CHORDS_INIT = Method.getMethod("void $initChordManager$()");
33   Method CHORDS_ADD_ASYNC = Method.getMethod("void addAsync(String, Object[])
         ");
34   Method CHORDS_ADD_SYNC = Method.getMethod("Object addSync(String, Object[])
         ");
35   Method CHORDS_RESULT = Method.getMethod("Object result(Class)");
36
37   String MANAGER_FIELD_NAME = "$ChordManager$";
38 }
```

Listagem A.14. jchords/util/Constants.java

A.2 Casos de Uso

A.2.1 Métodos Assíncronos

```
1  package usecases.async.java;
2
3  import static jchords.util.Utils.delay;
4
5  public class Example1 extends Thread {
6    public void run() {
7      delay(1000, "run");
8    }
9
10   public static void main(String[] args) {
11     System.out.println("inicio");
12     for (int index = 0; index < 5; ++index) {
13       new Example1().start();
14     }
15     System.out.println("fim");
16   }
17 }
```

Listagem A.15. usecases/async/java/Example1.java

```
1  package usecases.async.java;
2
3  import static jchords.util.Utils.delay;
```

```
4
5  public class Example2 {
6    public void foo(int x) {
7      delay(1000, "foo " + x);
8    }
9
10   public void bar() {
11     delay(1000, "bar");
12   }
13
14   public static void main(String[] args) {
15     System.out.println("inicio");
16     final Example2  example = new Example2();
17     for (int index = 0; index < 5; ++index) {
18       final int local = index;
19       new Thread() {
20         public void run() {
21           example.foo(local);
22         }
23       }.start();
24       new Thread() {
25         public void run() {
26           example.bar();
27         }
28       }.start();
29     }
30     System.out.println("fim");
31   }
32 }
```

Listagem A.16. usecases/async/java/Example2.java

```
1  package usecases.async.jchords;
2
3  import static jchords.util.Utils.delay;
4  import jchords.annotations.Async;
5
6  public class Example1 {
7    @Async public void foo() {
8      delay(1000, "foo");
9    }
```

```
10
11   public static void main(String[] args) {
12     System.out.println("inicio");
13     Example1 example = new Example1();
14     for (int index = 0; index < 5; ++index) {
15       example.foo();
16     }
17     System.out.println("fim");
18   }
19 }
```

Listagem A.17. usecases/async/jchords/Example1.java

```
1  package usecases.async.jchords;
2
3  import static jchords.util.Utils.delay;
4  import jchords.annotations.Async;
5
6  public class Example2 {
7    @Async public void foo(int x) {
8      delay(1000, "foo " + x);
9    }
10
11   @Async public void bar() {
12     delay(1000, "bar");
13   }
14
15   public static void main(String[] args) {
16     System.out.println("inicio");
17     Example2 example = new Example2();
18     for (int index = 0; index < 5; ++index) {
19       example.foo(index);
20       example.bar();
21     }
22     System.out.println("fim");
23   }
24 }
```

Listagem A.18. usecases/async/jchords/Example2.java

A.2.2 Jantar dos Filósofos

```
1  package usecases.diningphilosophers.higienic.jchords;
2
3  import jchords.annotations.*;
4
5  @Chorded public class Fork {
6    @Sync public void grab() {}
7    @Async public void release() {}
8
9    public Fork() {
10     release();
11   }
12
13   @Join({"void grab()", "void release()"}) public void grab_release() {}
14 }
```

Listagem A.19. usecases/diningphilosophers/higienic/jchords/Fork.java

```
1  package usecases.diningphilosophers.higienic.jchords;
2
3  import static jchords.util.Utils.delay;
4  import jchords.annotations.*;
5
6  @Chorded public class Phil {
7    private static int seed = 0;
8    public final int id = ++seed;
9
10   @Async public void requires(Phil phil) {}
11   @Async public void clean(Fork fork) {}
12   @Async public void dirty(Fork fork) {}
13   @Async public void request(Phil phil) {}
14   @Async public void hungry() {}
15   @Sync public void eating() {}
16
17   @Join({"void eating()", "void hungry()",
18     "void clean(usecases.diningphilosophers.higienic.jchords.Fork)",
19     "void clean(usecases.diningphilosophers.higienic.jchords.Fork)"})
20   public void eat_hungry(Fork left, Fork right) {
21     left.grab();
22     right.grab();
```

```
23      eat();
24      left.release();
25      right.release();
26      dirty(left);
27      dirty(right);
28    }
29
30    @Join({"void hungry()",
31      "void requires(usecases.diningphilosophers.higienic.jchords.Phil)"})
32    public void hungry_requires(Phil phil) {
33      hungry();
34      phil.request(this);
35    }
36
37    @Join({"void request(usecases.diningphilosophers.higienic.jchords.Phil)",
38      "void dirty(usecases.diningphilosophers.higienic.jchords.Fork)"})
39    public void request_dirty(Phil phil, Fork fork) {
40      requires(phil);
41      phil.clean(fork);
42    }
43
44    @Async public void execute() {
45      while (true) {
46        think();
47        hungry();
48        eating();
49      }
50    }
51
52    private void eat() {
53      delay(5000, "Filósofo " + id + " comendo");
54    }
55
56    private void think() {
57      delay(1000, "Filósofo " + id + " pensando");
58    }
59  }
```

Listagem A.20. usecases/diningphilosophers/higienic/jchords/Phil.java

```
1  package usecases.diningphilosophers.higienic.jchords;
```

```
2
3  public class Runner {
4    private static final int SEATS = 5;
5
6    public static void main(String[] args) throws InterruptedException {
7      System.out.println("inicio");
8      Phil[] phils = new Phil[SEATS];
9
10     for(int index = 0; index < SEATS; ++index) {
11       phils[index] = new Phil();
12       if(index != 0) {
13         phils[index].requires(phils[index-1]);
14       } else {
15         phils[index].dirty(new Fork());
16       }
17       if(index != SEATS - 1) {
18         phils[index].dirty(new Fork());
19       } else {
20         phils[index].requires(phils[0]);
21       }
22     }
23     for(int index = 0; index < SEATS; ++index) {
24       phils[index].execute();
25     }
26     System.out.println("fim");
27   }
28 }
```

Listagem A.21. usecases/diningphilosophers/higienic/jchords/Runner.java

```
1  package usecases.diningphilosophers.waiter.java;
2
3  import jchords.annotations.*;
4
5  @Chorded public class Fork {
6    @Sync public void grab() {}
7    @Async public void release() {}
8
9    public Fork() {
10     release();
11   }
```

```
12
13   @Join({"void grab()", "void release()"}) public void grab_release() {}
14 }
```

Listagem A.22. usecases/diningphilosophers/waiter/java/Fork.java

```
1  package usecases.diningphilosophers.waiter.java;
2
3  import static jchords.util.Utils.delay;
4
5  public class Phil extends Thread {
6    private static int seed = 0;
7    public final int id = ++seed;
8    private final Waiter waiter;
9    private final Fork left;
10   private final Fork right;
11
12   public Phil(Waiter waiter, Fork left, Fork right) {
13     this.waiter = waiter;
14     this.left = left;
15     this.right = right;
16   }
17
18   public void run() {
19     try {
20       while (true) {
21         think();
22         waiter.enter();
23         left.grab();
24         right.grab();
25         eat();
26         left.release();
27         right.release();
28         waiter.leave();
29       }
30     } catch (InterruptedException ex) {
31       ex.printStackTrace();
32     }
33   }
34
35   private void eat() {
```

```
36      delay(1000, "Filósofo " + id + " comendo");
37    }
38
39    private void think() {
40      delay(5000, "Filósofo " + id + " pensando");
41    }
42  }
```

Listagem A.23. usecases/diningphilosophers/waiter/java/Phil.java

```
1   package usecases.diningphilosophers.waiter.java;
2
3   public class Runner {
4     public static final int SEATS = 5;
5
6     public static void main(String[] args) throws InterruptedException {
7       System.out.println("inicio");
8       Waiter waiter = new Waiter();
9       Fork[] forks = new Fork[SEATS];
10      for(int index = 0; index < SEATS; ++index) {
11        forks[index] = new Fork();
12      }
13      for(int index = 0; index < SEATS; ++index) {
14        new Phil(waiter, forks[index], forks[(index + 1) % SEATS]).start();
15      }
16      System.out.println("fim");
17    }
18  }
```

Listagem A.24. usecases/diningphilosophers/waiter/java/Runner.java

```
1   package usecases.diningphilosophers.waiter.java;
2
3   public class Waiter {
4     private int count = Runner.SEATS - 1;
5
6     public synchronized void enter() throws InterruptedException {
7       while(count > 0) {
8         wait();
9       }
10      --count;
```

```
11    }
12
13    public synchronized void leave() {
14      ++count;
15      notifyAll();
16    }
17  }
```

Listagem A.25. usecases/diningphilosophers/waiter/java/Waiter.java

```
1  package usecases.diningphilosophers.waiter.jchords;
2
3  import jchords.annotations.*;
4
5  public class Fork {
6    @Sync public void grab() {}
7    @Async public void release() {}
8
9    public Fork() {
10     release();
11   }
12
13   @Join({"void grab()", "void release()"}) public void grab_release() {}
14 }
```

Listagem A.26. usecases/diningphilosophers/waiter/jchords/Fork.java

```
1  package usecases.diningphilosophers.waiter.jchords;
2
3  import static jchords.util.Utils.delay;
4  import jchords.annotations.Async;
5
6  public class Phil {
7    private static int seed = 0;
8    public final int id = ++seed;
9    private final Waiter waiter;
10   private final Fork left;
11   private final Fork right;
12
13   public Phil(Waiter waiter, Fork left, Fork right) {
14     this.waiter = waiter;
```

```
15      this.left = left;
16      this.right = right;
17    }
18
19    @Async public void execute() {
20      while (true) {
21        think();
22        waiter.enter();
23        left.grab();
24        right.grab();
25        eat();
26        left.release();
27        right.release();
28        waiter.leave();
29      }
30    }
31
32    private void eat() {
33      delay(1000, "Filósofo " + id + " comendo");
34    }
35
36    private void think() {
37      delay(5000, "Filósofo " + id + " pensando");
38    }
39  }
```

Listagem A.27. usecases/diningphilosophers/waiter/jchords/Phil.java

```
1   package usecases.diningphilosophers.waiter.jchords;
2
3   public class Runner {
4     public static final int SEATS = 5;
5
6     public static void main(String[] args) throws InterruptedException {
7       System.out.println("inicio");
8       Waiter waiter = new Waiter();
9       Fork[] forks = new Fork[SEATS];
10      for(int index = 0; index < SEATS; ++index) {
11        forks[index] = new Fork();
12      }
13      for(int index = 0; index < SEATS; ++index) {
```

```
14        new Phil(waiter, forks[index], forks[(index + 1) % SEATS]).execute();
15      }
16      System.out.println("fim");
17    }
18  }
```

Listagem A.28. usecases/diningphilosophers/waiter/jchords/Runner.java

```
1  package usecases.diningphilosophers.waiter.jchords;
2
3  import jchords.annotations.*;
4
5  @Chorded public class Waiter {
6    @Sync public void enter() {}
7    @Async public void leave() {}
8
9    public Waiter() {
10     for (int index = 0; index < Runner.SEATS - 1; ++index) {
11       leave();
12     }
13   }
14
15   @Join({"void enter()", "void leave()"}) public void enter_leave() {}
16 }
```

Listagem A.29. usecases/diningphilosophers/waiter/jchords/Waiter.java

A.2.3 Produtor Consumidor

```
1  package usecases.producerconsumer.java;
2
3  public class Buffer {
4    private int count = 0;
5    private int[] resources = new int[Runner.BUFFER_SIZE];
6
7    public synchronized void put(int res) throws InterruptedException {
8      while (count == resources.length) {
9        wait();
10     }
11     resources[count++] = res;
12     notifyAll();
```

```
13    }
14
15    public synchronized int get() throws InterruptedException {
16      while (count == 0) {
17        wait();
18      }
19      int res = resources[--count];
20      notifyAll();
21      return res;
22    }
23  }
```

Listagem A.30. usecases/producerconsumer/java/Buffer.java

```
1  package usecases.producerconsumer.java;
2
3  import static jchords.util.Utils.delay;
4
5  public class Consumer extends Thread {
6    private static int seed = 0;
7    private final int id = ++seed;
8    private final Buffer buffer;
9
10   public Consumer(Buffer buffer) {
11     this.buffer = buffer;
12   }
13
14   public void run() {
15     try {
16       while (true) {
17         consume(buffer.get());
18       }
19     } catch (InterruptedException ex) {
20       ex.printStackTrace();
21     }
22   }
23
24   private void consume(int resource) {
25     delay(0, 10000, "Consumidor " + id + " consumindo " + resource);
26   }
27 }
```

Listagem A.31. usecases/producerconsumer/java/Consumer.java

```
1  package usecases.producerconsumer.java;
2
3  import static jchords.util.Utils.delay;
4
5  public class Producer extends Thread {
6    private static int seed = 0;
7    private final int id = ++seed;
8    private final Buffer buffer;
9
10   public Producer(Buffer buffer) {
11     this.buffer = buffer;
12   }
13
14   public void run() {
15     try {
16       while (true) {
17         buffer.put(produce());
18       }
19     } catch (InterruptedException ex) {
20       ex.printStackTrace();
21     }
22   }
23
24   private int produce() throws InterruptedException {
25     int resource = (int) Math.random() * 100000000;
26     delay(0, 10000, "Produtor " + id + " produzindo " + resource);
27     return resource;
28   }
29 }
```

Listagem A.32. usecases/producerconsumer/java/Producer.java

```
1  package usecases.producerconsumer.java;
2
3  public class Runner {
4    public static final int PRODUCERS = 5;
5    public static final int CONSUMERS = 8;
6    public static final int BUFFER_SIZE = 10;
```

```
7
8   public static void main(String[] args) {
9     System.out.println("inicio");
10    Buffer buffer = new Buffer();
11    for(int index = 0; index < PRODUCERS; ++index) {
12      new Producer(buffer).start();
13    }
14    for(int index = 0; index < CONSUMERS; ++index) {
15      new Consumer(buffer).start();
16    }
17    System.out.println("fim");
18  }
19 }
```

Listagem A.33. usecases/producerconsumer/java/Runner.java

```
1  package usecases.producerconsumer.jchords;
2
3  import jchords.Chords;
4  import jchords.annotations.*;
5
6  @Chorded public class Buffer {
7    @Async public void value(int val) {}
8    @Async public void free() {}
9    @Sync public void put(int val) {}
10   @Sync public int get() { return Chords.result(int.class); }
11
12   public Buffer() {
13     for(int index = 0 ; index < Runner.BUFFER_SIZE; ++index) {
14       free();
15     }
16   }
17
18   @Join({"int get()", "void value(int)"}) public int get_value(int res) {
19     free();
20     return res;
21   }
22
23   @Join({"void put(int)", "void free()"}) public void put_free(int res) {
24     value(res);
25   }
```

```
26 }
```

Listagem A.34. usecases/producerconsumer/jchords/Buffer.java

```
1  package usecases.producerconsumer.jchords;
2
3  import static jchords.util.Utils.delay;
4  import jchords.annotations.Async;
5
6  public class Consumer {
7    private static int seed = 0;
8    private final Buffer buffer;
9    private final int id = ++seed;
10
11   public Consumer(Buffer buffer) {
12     this.buffer = buffer;
13   }
14
15   @Async public void execute() {
16     while (true) {
17       consume(buffer.get());
18     }
19   }
20
21   private void consume(int resource) {
22     delay(0, 10000, "Consumidor " + id + " consumindo " + resource);
23   }
24 }
```

Listagem A.35. usecases/producerconsumer/jchords/Consumer.java

```
1  package usecases.producerconsumer.jchords;
2
3  import static jchords.util.Utils.delay;
4  import jchords.annotations.Async;
5
6  public class Producer {
7    private static int seed = 0;
8    private final int id = ++seed;
9    private int resource = id * 100000;
10   private final Buffer buffer;
```

```
11
12    public Producer(Buffer buffer) {
13      this.buffer = buffer;
14    }
15
16    @Async public void execute() {
17      while(true) {
18        buffer.put(produce());
19      }
20    }
21
22    private int produce() {
23      delay(0, 10000, "Produtor " + id + " produzindo " + resource++);
24      return resource;
25    }
26
27  }
```

Listagem A.36. usecases/producerconsumer/jchords/Producer.java

```
1  package usecases.producerconsumer.jchords;
2
3  public class Runner {
4    private static final int PRODUCERS = 5;
5    private static final int CONSUMERS = 5;
6    public static final int BUFFER_SIZE = 10;
7
8    public static void main(String[] args) throws InterruptedException {
9      System.out.println("inicio");
10     Buffer buffer = new Buffer();
11     for(int index = 0; index < PRODUCERS; ++index) {
12       new Producer(buffer).execute();
13     }
14     for(int index = 0; index < CONSUMERS; ++index) {
15       new Consumer(buffer).execute();
16     }
17     System.out.println("fim");
18   }
19
20 }
```

Listagem A.37. usecases/producerconsumer/jchords/Runner.java

A.2.4 Papai Noel

```
1  package usecases.santaclaus.java;
2
3  import static jchords.util.Utils.delay;
4
5  class Elf extends Thread {
6    private static int seed = 0;
7    private int id = ++seed;
8
9    public void run() {
10     try {
11       while (true) {
12         working();
13         SantaClaus.Santa.ElvesGroup.Register();
14         SantaClaus.Santa.SantaMonitor.EnterOffice();
15         consulting();
16         SantaClaus.Santa.SantaMonitor.LeaveOffice();
17       }
18     } catch (InterruptedException e) {}
19   }
20
21   private void consulting() {
22     delay(0, 2000, "Elfo " + id + " consultando");
23   }
24
25   private void working() {
26     delay(0, 6000, "Elfo " + id + " trabalhando");
27   }
28 }
```

Listagem A.38. usecases/santaclaus/java/Elf.java

```
1  package usecases.santaclaus.java;
2
3  class Group {
4    private static final int Open = 0;
```

```
5   private static final int Closed = 1;
6   private int GroupSize;
7   private int GroupID;
8   private int Waiting = 0;
9   private int Entrance = Open;
10  private int ExitDoor = Closed;
11
12  Group(int Size, int ID) {
13    GroupID = ID;
14    GroupSize = Size;
15  }
16
17  synchronized void Register() throws InterruptedException {
18    while (Entrance == Closed) {
19      wait();
20    }
21    Waiting++;
22    if(Waiting < GroupSize) {
23      while (ExitDoor == Closed) {
24        wait();
25      }
26    } else {
27      Entrance = Closed;
28      ExitDoor = Open;
29      notifyAll();
30    }
31    Waiting--;
32    if(Waiting == 0) {
33      SantaClaus.Santa.SantaMonitor.Wake(GroupID);
34    }
35  }
36
37  synchronized void OpenDoor() {
38    Entrance = Open;
39    ExitDoor = Closed;
40    notifyAll();
41  }
42 }
```

Listagem A.39. usecases/santaclaus/java/Group.java

```
1  package usecases.santaclaus.java;
2
3  class Monitor {
4    private static final int Sleeping = 0;
5    private static final int Harnessing = 1;
6    private static final int ShowingIn = 2;
7    private static final int Delivering = 3;
8    private static final int Consulting = 4;
9    private static final int UnHarnessing = 5;
10   private static final int ShowingOut = 6;
11
12   private int ReindeerHarnessed = 0;
13   private int ElvesEntered = 0;
14   private int ReindeerTeam;
15   private int ElfTeam;
16   private int State = Sleeping;
17
18   Monitor(int RTeam, int ETeam) {
19     ReindeerTeam = RTeam;
20     ElfTeam = ETeam;
21   }
22
23   synchronized void Wake(int GroupID) throws InterruptedException {
24     while (State != Sleeping) {
25       wait();
26     }
27     if(GroupID == SantaClaus.REINDEER_ID) {
28       State = Harnessing;
29     } else {// GroupID == SantaClaus.ElvesID
30       State = ShowingIn;
31     }
32     notifyAll();
33   }
34
35   synchronized int Who() throws InterruptedException {
36     while ((State != Delivering) && (State != Consulting)) {
37       wait();
38     }
39     if(State == Delivering) {
```

```
40        return SantaClaus.REINDEER_ID;
41      } else {// State == Consulting
42        return SantaClaus.ELVES_ID;
43      }
44    }
45
46    synchronized void Reset(int GroupID) throws InterruptedException {
47      if(GroupID == SantaClaus.REINDEER_ID) {
48        State = UnHarnessing;
49      } else {// GroupID == SantaClaus.ElvesID
50        State = ShowingOut;
51      }
52      notifyAll();
53    }
54
55    synchronized void Harness() throws InterruptedException {
56      while (State != Harnessing) {
57        wait();
58      }
59      ReindeerHarnessed++;
60      if(ReindeerHarnessed == ReindeerTeam) {
61        State = Delivering;
62        notifyAll();
63      }
64    }
65
66    synchronized void UnHarness() throws InterruptedException {
67      while (State != UnHarnessing) {
68        wait();
69      }
70      ReindeerHarnessed--;
71      if(ReindeerHarnessed == 0) {
72        State = Sleeping;
73        SantaClaus.Santa.ReinsGroup.OpenDoor();
74        notifyAll();
75      }
76    }
77
78    synchronized void EnterOffice() throws InterruptedException {
79      while (State != ShowingIn) {
```

```
80        wait();
81      }
82      ElvesEntered++;
83      if(ElvesEntered == ElfTeam) {
84        State = Consulting;
85        notifyAll();
86      }
87    }
88
89    synchronized void LeaveOffice() throws InterruptedException {
90      while (State != ShowingOut) {
91        wait();
92      }
93      ElvesEntered--;
94      if(ElvesEntered == 0) {
95        State = Sleeping;
96        SantaClaus.Santa.ElvesGroup.OpenDoor();
97        notifyAll();
98      }
99    }
100
101 }
```

Listagem A.40. usecases/santaclaus/java/Monitor.java

```
1  package usecases.santaclaus.java;
2
3  import static jchords.util.Utils.delay;
4
5  class Reindeer extends Thread {
6    private static int seed = 0;
7    private int id = ++seed;
8
9    public void run() {
10     try {
11       while (true) {
12         vacationing();
13         SantaClaus.Santa.ReinsGroup.Register();
14         SantaClaus.Santa.SantaMonitor.Harness();
15         delivering();
16         SantaClaus.Santa.SantaMonitor.UnHarness();
```

```
17        }
18      } catch (InterruptedException e) {}
19    }
20
21    private void delivering() {
22      delay(0, 1000, "Rena " + id + " entregando brinquedos");
23    }
24
25    private void vacationing() {
26      delay(0, 10000, "Rena " + id + " de férias");
27    }
28
29  }
```

Listagem A.41. usecases/santaclaus/java/Reindeer.java

```
1  package usecases.santaclaus.java;
2
3  public class Runner {
4    private static final int REINDEER = 9;
5    private static final int ELVES = 10;
6
7    public static void main(String[] args) {
8      System.out.println("inicio");
9      for (int i = 0; i < REINDEER; i++) {
10       new Reindeer().start();
11     }
12     for (int i = 0; i < ELVES; i++) {
13       new Elf().start();
14     }
15     SantaClaus.Santa.start();
16     System.out.println("fim");
17   }
18 }
```

Listagem A.42. usecases/santaclaus/java/Runner.java

```
1  package usecases.santaclaus.java;
2
3  class SantaClaus extends Thread {
4    static final int REIN_TEAM = 9;
```

```
5   static final int ELF_TEAM = 3;
6
7   static final int REINDEER_ID = 1;
8   static final int ELVES_ID = 2;
9
10  public static SantaClaus Santa = new SantaClaus();
11  Monitor SantaMonitor = new Monitor(REIN_TEAM, ELF_TEAM);
12  public Group ReinsGroup = new Group(REIN_TEAM, REINDEER_ID);
13  public Group ElvesGroup = new Group(ELF_TEAM, ELVES_ID);
14
15  public void run() {
16    try {
17      int signalled;
18      while (true) {
19        sleeping();
20        signalled = SantaMonitor.Who();
21        if(signalled == REINDEER_ID) {
22          delivering();
23        } else if(signalled == ELVES_ID) {
24          consulting();
25        }
26        SantaMonitor.Reset(signalled);
27      }
28    } catch (InterruptedException e) {}
29  }
30
31  private void consulting() {
32    System.out.println("Papai Noel consultando");
33  }
34
35  private void delivering() {
36    System.out.println("Papai Noel entregando brinquedos");
37  }
38
39  private void sleeping() {
40    System.out.println("Papai Noel dormindo");
41  }
42 }
```

Listagem A.43. usecases/santaclaus/java/SantaClaus.java

```
1  package usecases.santaclaus.jchords;
2
3  import static jchords.util.Utils.delay;
4  import jchords.annotations.Async;
5
6  public class Elf {
7    private static int seed = 0;
8    private int id = ++seed;
9
10   @Async public void execute() {
11     while (true) {
12       working();
13       SantaClaus.INSTANCE.elf();
14       SantaClaus.INSTANCE.roomin.entry();
15       consulting();
16       SantaClaus.INSTANCE.roomout.entry();
17     }
18   }
19
20   private void consulting() {
21     delay(0, 2000, "Elfo " + id + " consultando");
22   }
23
24   private void working() {
25     delay(0, 6000, "Elfo " + id + " trabalhando");
26   }
27 }
```

Listagem A.44. usecases/santaclaus/jchords/Elf.java

```
1  package usecases.santaclaus.jchords;
2
3  import jchords.annotations.*;
4
5  @Chorded public class Group {
6    @Sync public void entry() {}
7    @Async public void tokens(int n) {}
8    @Sync public void waitt() {}
9    @Async public void allgone() {}
10
```

```
11   public void accept(int n) {
12     tokens(n);
13     waitt();
14   }
15
16   @Join({"void entry()", "void tokens(int)"})
17   public void entry_tokens(int n) {
18     if (--n == 0) {
19       allgone();
20     } else {
21       tokens(n);
22     }
23   }
24
25   @Join({"void waitt()", "void allgone()"}) public void wait_allgone() {}
26 }
```

Listagem A.45. usecases/santaclaus/jchords/Group.java

```
1  package usecases.santaclaus.jchords;
2
3  import static jchords.util.Utils.delay;
4  import jchords.annotations.Async;
5
6  public class Reindeer {
7    private static int seed = 0;
8    private int id = ++seed;
9
10   @Async public void execute() {
11     while (true) {
12       vacationing();
13       SantaClaus.INSTANCE.rein();
14       SantaClaus.INSTANCE.harness.entry();
15       delivering();
16       SantaClaus.INSTANCE.unharness.entry();
17     }
18   }
19
20   private void delivering() {
21     delay(0, 1000, "Rena " + id + " entregando brinquedos");
22   }
```

```
23
24   private void vacationing() {
25     delay(0, 10000, "Rena " + id + " de férias");
26   }
27 }
```

Listagem A.46. usecases/santaclaus/jchords/Reindeer.java

```
1  package usecases.santaclaus.jchords;
2
3  public class Runner {
4
5    private static final int REINDEER = 9;
6    private static final int ELVES = 10;
7
8    public static void main(String[] args) throws InterruptedException {
9      System.out.println("inicio");
10     for (int i = 0; i < REINDEER; i++) {
11       new Reindeer().execute();
12     }
13     for (int i = 0; i < ELVES; i++) {
14       new Elf().execute();
15     }
16     System.out.println("fim");
17   }
18 }
```

Listagem A.47. usecases/santaclaus/jchords/Runner.java

```
1  package usecases.santaclaus.jchords;
2
3  import jchords.annotations.*;
4
5  @Chorded public class SantaClaus {
6    public static final int REIN_TEAM = 9;
7    public static final int ELF_TEAM = 3;
8    public static SantaClaus INSTANCE = new SantaClaus();
9
10   public final Group harness = new Group();
11   public final Group unharness = new Group();
12   public final Group roomin = new Group();
```

```
13  public final Group roomout = new Group();
14
15  private SantaClaus() {
16    elves(0);
17    reins(0);
18    elvesfirst();
19    sleep();
20  }
21
22  @Async public void sleep() {
23    System.out.println("Papai Noel dormindo");
24  }
25  @Async public void elf() {}
26  @Async public void elves(int e) {}
27  @Async public void elvesready() {}
28  @Async public void elvesfirst() {}
29  @Async public void rein() {}
30  @Async public void reins(int i) {}
31  @Async public void reinsready() {}
32  @Async public void reinsfirst() {}
33
34  @Join({"void reinsfirst()", "void elvesfirst()"})
35  public void reinfirst_elvesfirst() {}
36
37  @Join({"void elf()", "void elves(int)"}) public void elf_elves(int e) {
38    if (++e == ELF_TEAM) {
39      elvesready();
40    } else {
41      elves(e);
42    }
43  }
44
45  @Join({"void rein()", "void reins(int)"}) public void rein_reins(int r) {
46    if (++r == REIN_TEAM) {
47      reinsfirst();
48      reinsready();
49    } else {
50      reins(r);
51    }
52  }
```

```
53
54   @Join({"void sleep()", "void reinsready()"})
55   public void sleep_reinsready() {
56     harness.accept(REIN_TEAM);
57     elvesfirst();
58     reins(0);
59     delivering();
60     unharness.accept(REIN_TEAM);
61     sleep();
62   }
63
64   @Join({"void sleep()", "void elvesready()", "void elvesfirst()"})
65   public void sleep_elvesready() {
66     elvesfirst();
67     roomin.accept(ELF_TEAM);
68     elves(0);
69     consulting();
70     roomout.accept(ELF_TEAM);
71     sleep();
72   }
73
74   private void consulting() {
75     System.out.println("Papai Noel consultando");
76   }
77
78   private void delivering() {
79     System.out.println("Papai Noel entregando brinquedos");
80   }
81 }
```

Listagem A.48. usecases/santaclaus/jchords/SantaClaus.java

A.2.5 Buffer Simples

```
1 package usecases.simplebuffer.jchords;
2
3 import jchords.Chords;
4 import jchords.annotations.*;
5
6 @Chorded public class Buffer {
```

```
7   @Async public void put(int var) {}
8   @Sync public int get() { return Chords.result(int.class); }
9   @Join({"int get()", "void put(int)"}) public int get_put(int var) {
10    return var;
11  }
12 }
```

Listagem A.49. usecases/simplebuffer/jchords/Buffer.java

```
1  package usecases.simplebuffer.jchords;
2
3  import jchords.*;
4  import jchords.annotations.*;
5  import jchords.annotations.Join;
6
7  @Chorded class BufferClass {
8    public BufferClass() {
9      $initChordManager$();
10   }
11
12   @Async public void Async$put(int value) {
13     $ChordManager$.addAsync("void put(int)", value);
14   }
15
16   public void put(int value) {
17     AsyncCall.invoke(BufferClass.class, "Async$put",
18       new Class<?>[] {int.class}, this, new Object[] {value});
19   }
20
21   @Sync public int get() {
22     return (Integer)$ChordManager$.addSync("int get()");
23   }
24
25   @Join({ "get()", "put(int)"}) public int get_put(int value) {
26     return value;
27   }
28
29   protected Chords $ChordManager$;
30
31   protected void $initChordManager$() {
32     $ChordManager$ = new Chords(this,
```

```
33        new JoinDescriptor("int get_put(int)", "int get()","void put(int)")
34      );
35    }
36  }
```

Listagem A.50. usecases/simplebuffer/jchords/BufferClass.java

Referências Bibliográficas

Benton, N. (2003). Jingle bells: Solving the santa claus problem in Polyphonic C#.

Benton, N.; Bierman, G.; Cardelli, L.; Meijer, E.; Russo, C. & Schulte, W. (2004). Cω.

Benton, N.; Cardelli, L. & Fournet, C. (2002). Modern concurrency abstractions for C#. In *ACM Transactions on Programming Languages and Systems*, pp. 415–440. Springer.

Benton, N.; Cardelli, L. & Fournet, C. (2007). Introduction to Polyphonic C#.

Berry, G. & Boudol, G. (1992). The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248.

Bruneton, E. (2007). *ASM 3.0: A Java bytecode engineering library.*

Buckley, A. (2004). JSR 175: A metadata facility for the Java programming language.

Butenhof, D. (1997). *Programming with POSIX Threads.* Professional Computing Series. Addison-Wesley.

Chandy, K. M. & Misra, J. (1984). The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6:632–646.

Chrysanthakopoulos, G. & Singh, S. (2005). An asynchronous messaging library for C#. In *SCOOL Conference Proceedings.*

Dijkstra, E. W. (1975). Guarded commands, non-determinacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457.

Drossopoulou, S.; Petrounias, A.; Buckley, A. & Eisenbach, S. (2006). SCHOOL: a small chorded object-oriented language. *Electronic Notes in Theoretical Computer Science*, 135(3):37–47.

Fournet, C.; Fessant, F. L.; Maranget, L. & Schmitt, A. (2002). JoCaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pp. 129–158. Springer.

Fournet, C. & Gonthier, G. (1996). The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pp. 372–385. ACM Press.

Fournet, C. & Gonthier, G. (2000). The join calculus: A language for distributed mobile programming. In *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pp. 268–332. Springer.

Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Resusable Object-Oriented Software*. Addison-Wesley Professional.

Gosling, J.; Joy, B.; Steele, G. & Bracha, G. (2005). *The Java(TM) Language Specification*. Java Series. Addison-Wesley Professional, 3rd edition edição.

Hoare, C. A. R. (1974). Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557.

Hoare, C. A. R. (1985). Communicating sequential processes. *Communications of the ACM*, 21:666–677.

Ingalls, D. (1981). Design principles behind smalltalk. *BYTE Magazine*, 6(8):286–298.

Itzstein, G. S. V. (2005). Introduction of high level concurrency semantics in object oriented languages. Master's thesis, University of South Australia.

Itzstein, G. S. V. & Jasiunas, M. (2003). On implementing high level concurrency in Java. In *Asia-Pacific Computer Systems Architecture Conference*, volume 2823 of *Lecture Notes in Computer Science*, pp. 151–165. Springer.

Itzstein, S. & Kearney, D. (2002). Applications of Join Java. In *Proceedings of the Seventh Asia-Pacific Computer Systems Architectures Conference (ACSAC2002*, pp. 37–46. Australian Computer Society, Inc.

Lee, E. (2006). The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley.

Liu, Y. (2008). Join: Asynchronous message based concurrency library based on Cω and join calculus.

Matsuoka, S. & Yonezawa, A. (1993). Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pp. 107–150. MIT Press.

Milner, R. (1980). A calculus of communicating systems. In *Proceedings CAAP 81, LNCS 112*, pp. 25–34. Springer-Verlag.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall.

Milner, R. (1992). Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141.

Odersky, M. (2000). Functional nets. In *Proceedings European Symposium on Programming*, number 1782 in LNCS, pp. 1–25. Springer Verlag.

Parrow, J. (2001). *Handbook of Process Algebra*, chapter An Introduction to the π-Calculus. Elsevier.

Petrounias, A.; Drossopoulou, S. & Eisenbach, S. (2008). A featherweight model for chorded languages. Technical report, Imperial College London.

Pierce, B. C. (1995). Foundational calculi for programming languages. In *Handbook of Computer Science and Engineering*. CRC Press.

Russo, C. (2006). The Joins concurrency library.

Sun Microsystems (2007). *JDK 5.0 Programmer Guides*. Sun Microsystems.

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software.

Trono, J. A. (1994). A new exercise in concurrency. *SIGCSE Bull*, 26(3):8–10.

Valente, M. T. O. (2002). *Mobilidade e Coordenação de Aplicações em Redes sem Fio*. PhD thesis, Universidade Federal de Minas Gerais, UFMG, Brasil.

Wood, T. & Eisenbach, S. (2004). A chorded compiler for Java.