6
Estudos - Provas de Conceito

Resumo

Neste capítulo são apresentados os resultados da aplicação da arquitetura proposta no desenvolvimento de um sistema de busca de publicações acadêmicas. Além disto, também utilizando ACCA, é discutida a implementação de alguns padrões de concorrência (Grand, 1998). Em conjunto com o problema exemplo apresentado no capítulo anterior, estes estudos têm por objetivo servir como prova de conceito da viabilidade do uso de ACCA.

6.1.
Sistema de Busca de Publicações Acadêmicas

Suponha o desenvolvimento de uma aplicação que reúna uma série de serviços de busca de publicações acadêmicas. Basicamente, esta aplicação precisa encadear um fluxo de consultas que serão feitas a cada serviço, contornando para isto, todas as restrições impostas pela integração das unidades de software existentes, e considerando para isto todas as interdependências e os requisitos necessários para a solução do problema.

6.1.1.
Características do Domínio

Cada vez mais componentes de software são desenvolvidos e distribuídos por uma vasta rede de informações e serviços. Neste sentido, artefatos de software locais precisam conviver com outras unidades locais ou remotamente distribuídas. Se por um lado esta característica oferece um maior número de alternativas, por outro, pode complicar o desenvolvimento de aplicações.

Neste cenário, a complexidade do desenvolvimento de uma aplicação pode ser atribuída a um maior número de unidades de software equivalentes ou similares e à necessidade de convivência de uma maior variedade de tecnologias de componentes em uma mesma solução. Outro fator característico do domínio em discussão é a velocidade com que requisitos evoluem, novas necessidades ou

possibilidades surgem a cada momento, dado o dinamismo desta nova realidade de desenvolvimento de aplicações.

6.1.2. Cenários de Uso dos Componentes: Tecnologias de Componentes Utilizadas

Diversos serviços de busca existem, cada um utilizando uma tecnologia diferente. Neste estudo de caso, *Web Services* distribuídos pela Internet precisam conviver com componentes de buscas tradicionais acessando bases de dados locais ou remotamente distribuídas. Cada tecnologia impõe protocolos de comunicação e protocolos de transporte que precisam ser integrados na geração de uma solução.

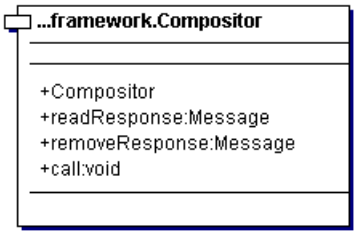

Figura 53 – Métodos presentes em uma composição

Analisando este problema é possível observar que uma composição deve apresentar três métodos básicos: *call* (submete uma requisição de consulta), o método *readResponse* (consulta a resposta de uma requisição de consulta) e o método *removeResponse* (retira a resposta a uma requisição de consulta). Todos estes métodos estão implementados pela classe *Composition* do *framework* de composição apresentado (Figura 53).

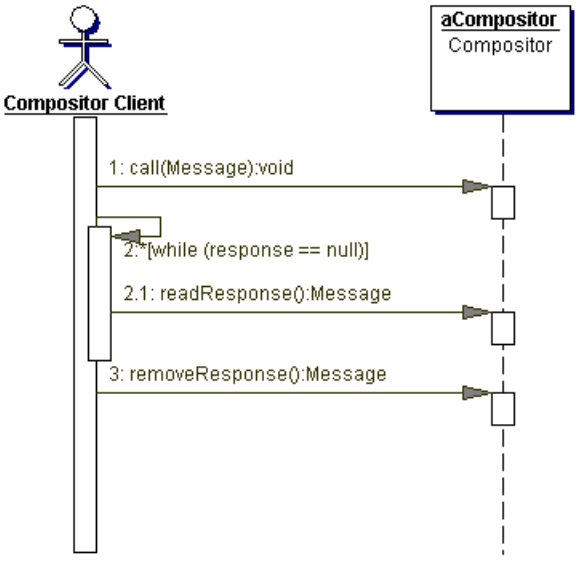

Figura 54 – Seqüência de utilização de instância de *Compositor*

Um cenário de uso (Figura 54) destes serviços de busca é: (1) submeter uma requisição de consulta por meio do método *call*, e periodicamente (2) verificar a disponibilidade de resposta a consulta efetuada por meio do método *readResponse,* e finalmente, quando necessário, (3) a resposta pode ser retirada do componente por meio do método *removeResponse*.

6.1.3. Cenários de Uso dos Componentes: Políticas de Coordenação

Os requisitos de um problema e as interdependências entre os serviços de uma solução geram um amplo conjunto de políticas de utilização dos serviços de consultas disponíveis. Exemplos de políticas variam muito, incluindo políticas simples que efetuam buscas seqüenciais em todas as composições existentes, políticas que privilegiam a proximidade da fonte de informações para determinar a seqüência de um cenário de uso, ou até políticas que restringem o tempo máximo de resposta de uma consulta efetuada por um usuário, interrompendo se for o caso a continuação do processo, além de diversas e variadas alternativas que podem ser elaboradas.

Abaixo, são descritas algumas regras que foram aplicadas ao desenvolvimento dos cenários de uso dos serviços de busca de publicações acadêmicas. Seu uso pode ser alternativo ou aditivo. O uso aditivo indica a possibilidade de convivência entre o conjunto de regras estipulado pelas políticas envolvidas. O uso alternativo indica a existência de opções equivalentes que podem ser utilizadas de acordo com uma necessidade que esteja sendo atendida.

Suponha a existência dos serviços de busca A, B, C e Z. Por exemplo, com relação à seqüência com que estes serviços de busca são encadeados: pode ser necessário estabelecer que o fluxo A→B→C deve ser seguido dada a proximidade das fontes de informações; ou ainda que fluxo B→A→C deve ser seguido dado o preço cobrado pelo uso das informações. Além disto, pode ser necessário seguir o fluxo C→B→A dada a qualidade das fontes de informações; e por fim, pode ser necessário redirecionar as chamadas feitas ao componente B ao serviço equivalente Z.

Como é possível observar, as políticas A→B→C, B→A→C ou C→B→A são alternativas, enquanto que o redirecionamento descrito pela última regra pode

ser aplicado em conjunto com as demais políticas, desde que esta política não interfira negativamente nas regras já estabelecidas.

Ainda com relação ao processo de coleta de informações, é possível estabelecer uma regra onde esteja estabelecido que se já existe resultado de uma das fontes de informações, não é necessário prosseguir com a busca nos demais serviços.

Além disto, é possível estabelecer uma política que indique a existência de uma estrutura de *caching* utilizada para otimizar o acesso a serviços de consulta. Regras desta política indicam que caso a informação desejada esteja disponível em *cache*, ela deve ser fornecida pelo *cache*, não sendo necessário o acesso ao serviço original. Caso contrário, acesse o serviço e sempre mantenha o *cache* atualizado.

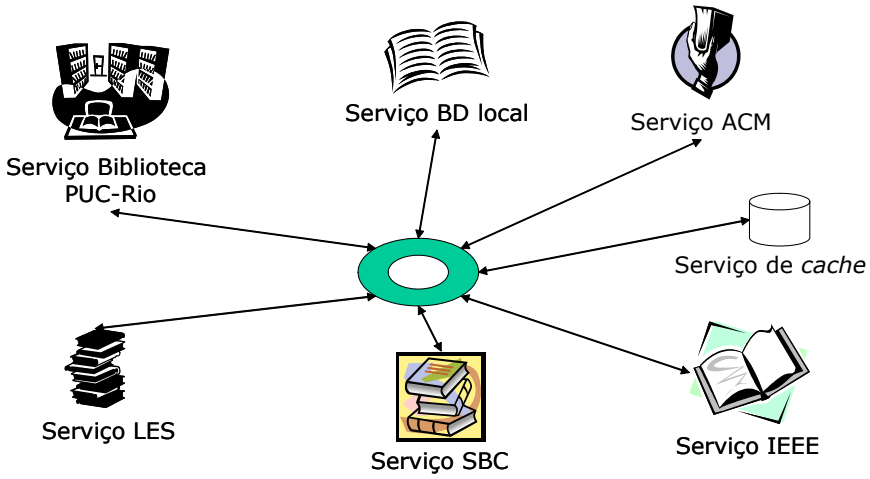

Figura 55 – Serviços de busca de publicações acadêmicas

6.1.4. Especificação do Sistema em ACCA

Neste cenário ilustrado (Figura 55), existem diversas fontes de informações, podendo cada uma destas disponibilizar um serviço seguindo uma tecnologia específica. No repositório da camada Artefatos de Software estão catalogados e descritos todos os serviços disponíveis e acessíveis para a construção de uma composição. Exemplos de serviços de busca que podem ser utilizados são: um serviço para acesso a uma base de dados local, um serviço de busca para acessar a base de publicações do LES, um serviço de busca para acessar a base de publicações acadêmicas da PUC-Rio, outros serviços de busca distribuídos remotamente em instituições privadas como a ACM (ACM, 2003) ou IEEE

(IEEE, 2003), ou mesmo a procura complementar em serviços de busca tradicionais como o Google (Google, 2003) ou o *Computer Science Search* da Elsevier (Elsevier, 2003). Além destes serviços de busca, existe um serviço de *cache* criado para otimizar a obtenção de informações a partir dos demais serviços.

Na camada de composição de ACCA, diversos artefatos existentes estão compostos em unidades de construção passíveis de integração. Um critério para se determinar que composições são relevantes é parte do projeto da solução. Para isto, o engenheiro de software deve analisar todos os requisitos funcionais e não funcionais da solução; deve transcrever todas as regras e interdependências que, por ventura, relacionam as unidades de software; e, a partir do levantamento destas unidades, algum critério deve ser determinado para a geração de composições.

É importante lembrar que os grupamentos em composições devem ser feitos caso os relacionamentos entre as unidades sejam relativamente estáveis. Caso contrário, o estabelecimento de interdependências deve ser feito via regras na camada de coordenação.

Um exemplo de critério para se determinar que unidades compõem grupamentos é a relevância de um serviço em função da equivalência de fontes de dados. Neste caso serviços de relevância equivalente, como IEEE e ACM, fazem parte de uma mesma composição (Figura 56).

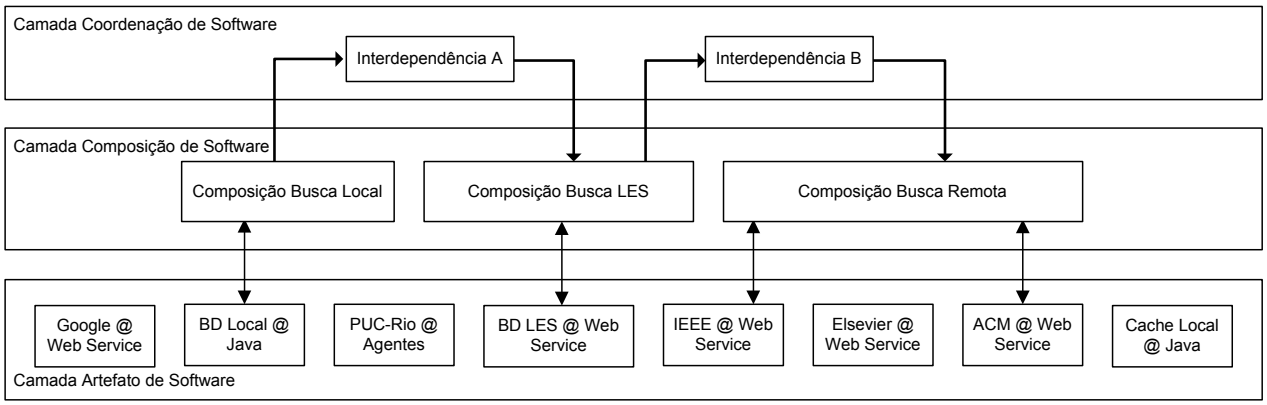

Figura 56 – Ilustração conceitual da realização das camadas de ACCA

Por fim, é possível ainda optar por um critério em que cada componente gera uma composição, e estas composições participam do processo de integração tendo as suas regras e suas interdependências impostas exclusivamente pela camada de coordenação.

Neste momento, pode ser necessário estender o *framework* de composição proposto para atender qualquer necessidade existente de integração. Estas necessidades variam desde a implementação de protocolos de transporte e comunicação, até a integração semântica das informações trocadas entre os componentes.

A partir das composições já estabelecidas, é possível determinar suas políticas de coordenação. Estas políticas são especificadas por meio da explicitação de interdependências entre as composições e são realizadas na camada de coordenação de ACCA. As regras existentes são convertidas em contratos de coordenação, que por sua vez, interagem com as composições criadas para a execução composta e coordenada dos serviços de busca e de *cache*.

6.1.5. Realização do Sistema em ACCA

Primeiramente, são descritos os contratos desenvolvidos para a coordenação de composições nesta seção. Além disto, é descrita a instância do *framework* de composição gerada para este estudo de caso.

Com relação à realização da camada de Artefatos de Software neste estudo, além do CVS (CVS, 2002) utilizado como catálogo de artefatos locais, é utilizado um catálogo manual contendo referências para todos os serviços passíveis de utilização. Este catálogo manual é um arquivo contendo a descrição dos serviços distribuídos e informações úteis para a sua localização.

6.1.5.1. Contratos de Coordenação

O primeiro contrato (Figura 57, Figura 58, Figura 59) descreve o processo de redirecionamento de uma chamada a um serviço equivalente. Para isto, o contrato de coordenação define regras para a interceptação dos métodos *call*, *readResponse*, e *removeResponse* do serviço *source* e os redireciona para o serviço *destiny*.

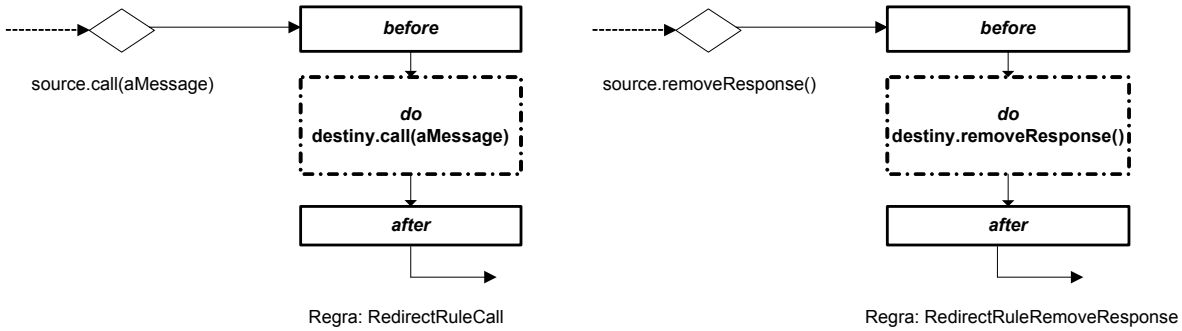

Figura 57 – Modelo de coordenação para o redirecionar chamadas (I//II)

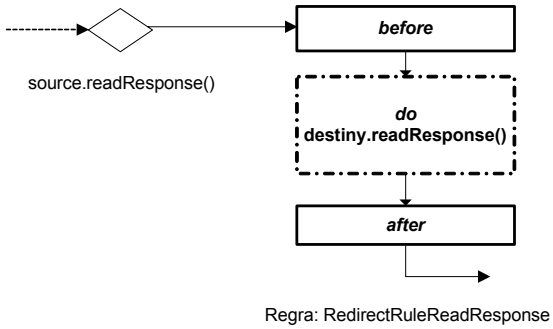

Figura 58 – Modelo de coordenação para o redirecionar chamadas (II/II)

```
contract ServiceRedirect
    participants
        source:Compositor;
        destiny:Compositor;

    coordination
    RedirectRuleCall:
        when *->> source.call(aMessage)
        do {
            destiny.call(aMessage);
        };

    RedirectRuleRemoveResponse:
        when *->> source.removeResponse()
        do {
            destiny.removeReponse();
        };

    RedirectRuleReadResponse:
        when *->> source.readResponse()
        do {
            destiny.readReponse();
        };
end contract //ServiceRedirect
```

Figura 59 – Exemplo de interceptação e redirecionamento de chamada a serviços

O segundo contrato (Figura 60, Figura 61) descreve o encadeamento de três serviços de busca. Durante o início deste processo, neste cenário de uso é acrescentada uma informação, correspondente ao número da transação equivalente, à mensagem de busca de informações. Porventura, esta informação pode ser utilizada por outro serviço para participar da solução.

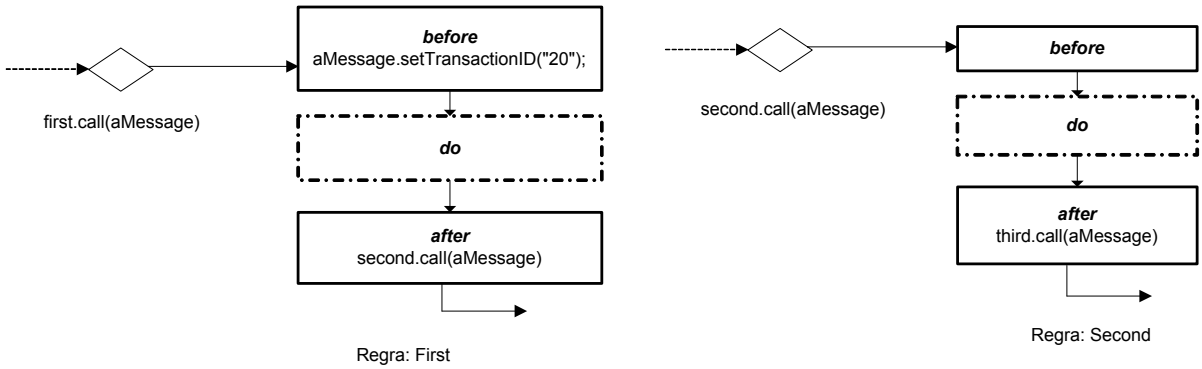

Figura 60 – Modelo de coordenação de encadeamento de serviços

```
contract ThreeSequenceService
    participants
        first:Compositor;
        second:Compositor;
        third:Compositor;

    coordination
    First:
       when *->> first.call(aMessage) && (aMessage != null)
            before {
                aMessage.setTransactionID("20"); }
            after {
                second.call(aMessage);
            };
    Second:
       when *->> second.call(aMessage) && (aMessage != null)
             after{
                third.call(aMessage);
             };
end contract //ThreeSequenceService
```

Figura 61 – Encadeamento de serviços de busca

O terceiro contrato (Figura 62, Figura 63) descreve parte do processo de atualização do *cache* de informações. Neste contrato, após alguém disparar um processo de consulta, a resposta do serviço origem das informações será consultada e o resultado da consulta será armazenado no serviço de *cache* criado especificamente para esta atividade.

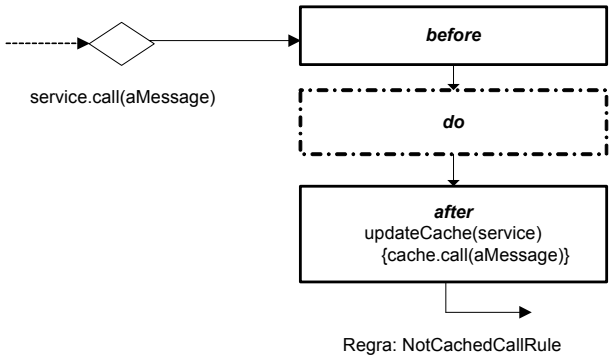

Figura 62 – Modelo de coordenação de atualização de *cache* de informações

```
contract UpdateCacheService
```

```
    participants
        service:Compositor;
        cache:Compositor;

    operations
    private void updateCache(Compositor aService) {
      Message message = aService.readResponse();
      if (message != null) {

          Query aQuery = (Query) message.getContent();
          Query to = new Query(aQuery.getQuery(), "CACHED");
          to.setResponse(aQuery.getResponse());
          Message cacheMessage = new Message(to,"100");
          cache.call(cacheMessage);}

    }

    coordination
    NotCachedCallRule:
      when *->> service.call(aMessage)
        after {
          updateCache(service);
        };

end contract //UpdateCacheService
```

Figura 63 – Atualização de *cache* de serviço de busca

Ainda sobre o *caching* de informações, o quarto contrato (Figura 64, Figura 65) foi criado para interceptar uma chamada a um serviço de busca, caso haja informação em *cache* sobre a consulta em andamento, esta requisição não será repassada ao serviço de consulta e o resultado será retornado efetivamente pelo serviço de *cache*. Neste contrato existe uma estrutura intermediária utilizada para armazenar as mensagens obtidas a partir do *cache*. Esta foi uma decisão de projeto tomada para retornar as respostas oriundas do *cache*.

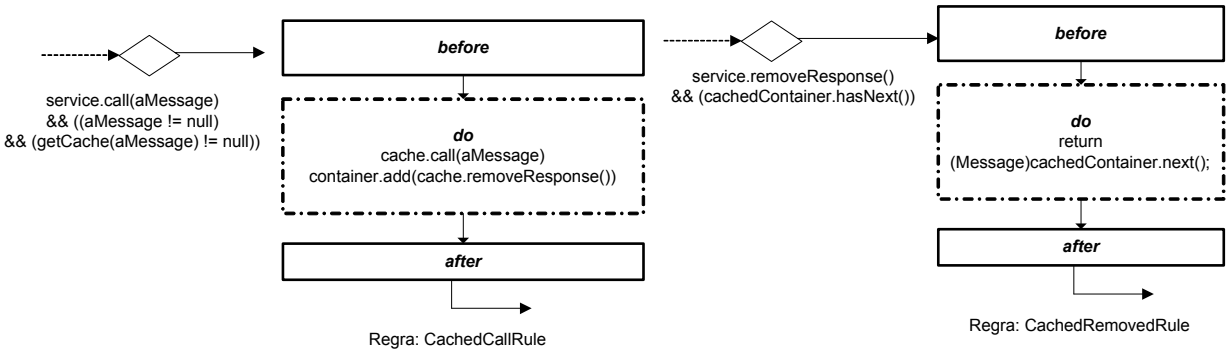

Figura 64 – Modelo de coordenação de acesso ao *cache*

```
contract CacheService
    participants
        service:Compositor;
        cache:Compositor;
    attributes
    private MessageContainer cachedContainer =
                                    new MessageContainer();
    operations
    private Message getCache(Message message)
    {
      if (message != null){
          String query = (String)message.getContent();
          Query aQuery =
          new Query(query,"services.providers.Cache");
          aQuery.setProvider("services.providers.Cache");
          Message cacheMessage = new Message(aQuery,"100");
          cache.call(cacheMessage);
       }
     return cache.readResponse();
     }
    coordination
    CachedCallRule:
        when *->> service.call(aMessage)
                  && ((aMessage != null)
                  && (getCache(aMessage) != null))
        do {
              Message resposta = cache.removeResponse();
              Query respQuery;
              if (resposta != null){
                  respQuery = (Query) resposta.getContent();}
              cachedContainer.add(resposta);
       };
    CachedRemovedRule:
        when *->> service.removeResponse()
                  && (cachedContainer.hasNext())
        do{
             return (Message)cachedContainer.next();
        };
end contract //CacheService
```

Figura 65 – Acesso ao *cache*: obtenção de informações

O quinto contrato (Figura 66, Figura 67) determina que a ordem de execução de serviços seja feita de acordo com a ordem em que os serviços correspondentes aos parâmetros são recebidos durante o processo de instanciação do contrato. Isto é, a primeira composição recebida é executada, seguida pela execução da segunda composição, e por fim, é executada a terceira composição. Esta seqüência de execuções é disparada quando ocorre uma primeira chamada a um método do primeiro serviço. Caso não se tenha obtido resposta deste serviço, é disparada uma chamada ao segundo serviço, e assim sucessivamente. Esta política pode ser utilizada caso se deseje implementar o encadeamento de serviços a partir de algum esquema de proximidade de serviço, isto é, as consultas são executadas de acordo com a proximidade da fonte de informações.

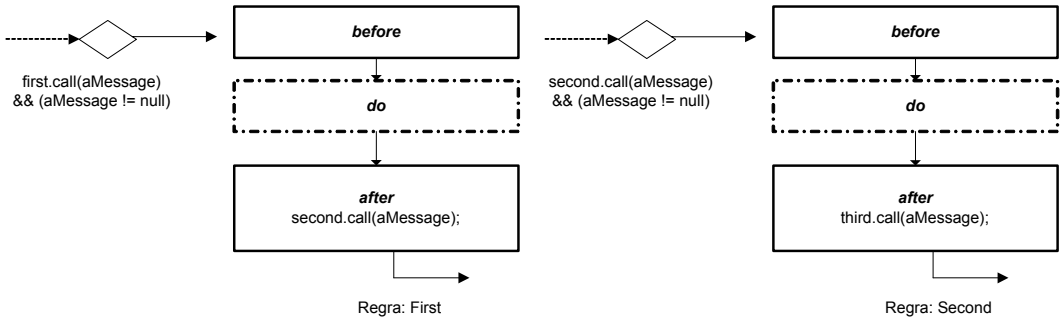

Figura 66 – Modelo de coordenação de seqüência de serviços

```
contract LocalPrioritySequence
    participants
        first:Compositor;
        second:Compositor;
        third:Compositor;
    operations
    private Message getResult(Compositor comp){
        Message received = comp.readResponse();
        return received; }
    coordination
    First:
       when *->> first.call(aMessage) && (aMessage != null)
           after{
                Message answer = getResult(first);
                if (answer == null){
                   second.call(aMessage);
                }
           };
    Second:
       when *->> second.call(aMessage) && (aMessage != null)
           after{
                Message answer = getResult(second);
                if (answer == null){
                    third.call(aMessage);
                }
           };
end contract
```

Figura 67 – Estabelecimento de prioridade para a obtenção das informações

6.1.5.2. Instância do *Framework* de Composição

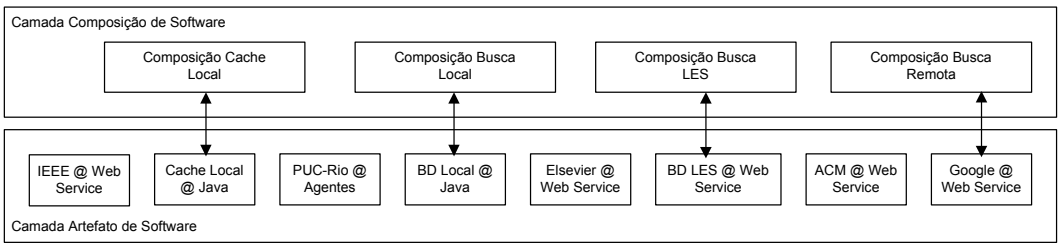

Figura 68 – Ilustração de relacionamento entre as composições e os serviços

Na realização da camada de composição do estudo de caso (Figura 68), cada tecnologia de serviço de busca é representada por uma única composição. Existem duas composições básicas: a composição referente a serviços implementados

utilizando *Web Services* e a composição referente a serviços locais implementados com Java.

Com relação à instância do *framework* de composição, é necessário implementar a interface *CompositionTransport* para atender ao protocolo de transporte HTTP. Além disto, a partir de outra implementação desta mesma interface, é viabilizada a conexão com artefatos locais. Estas implementações permitem que haja interação entre as composições e os artefatos de software que oferecem os respectivos serviços.

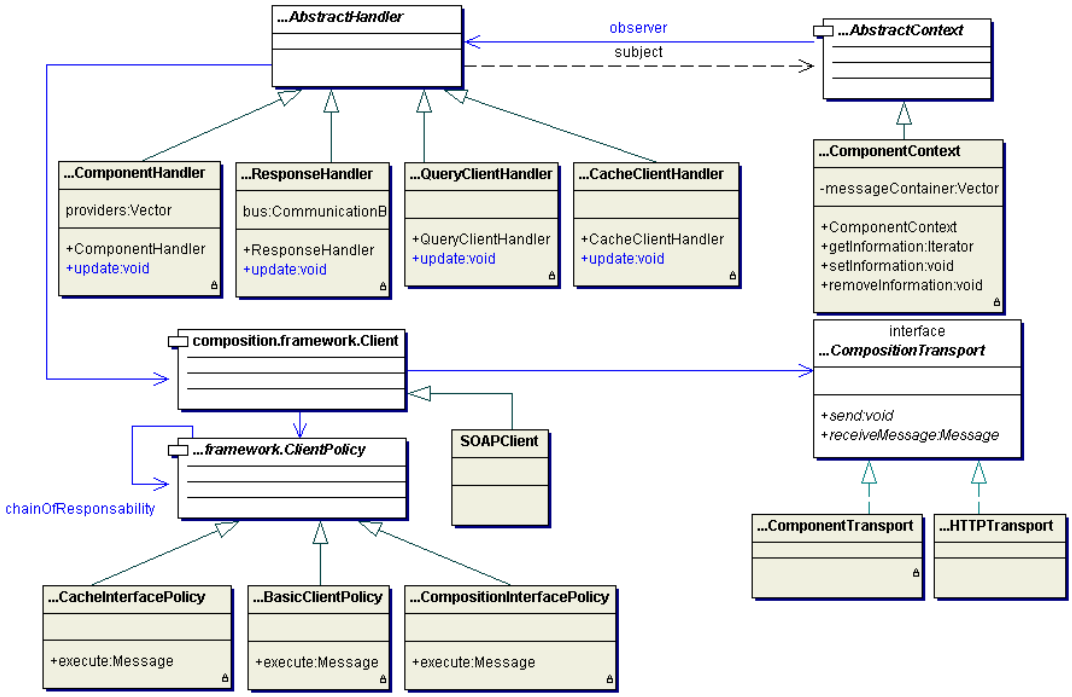

Figura 69 – Instância do *framework*: transporte, comunicação e *Context-Handler*

Para o tratamento de mensagens recebidas ao longo do processo de composição são implementadas sub-classes de *AbstractHandler,* chamadas de *ComponentHandler, ResponseHandler, QueryClientHandler e CacheHandler.* Estas classes são responsáveis por encapsular o processo de chamada a serviços.

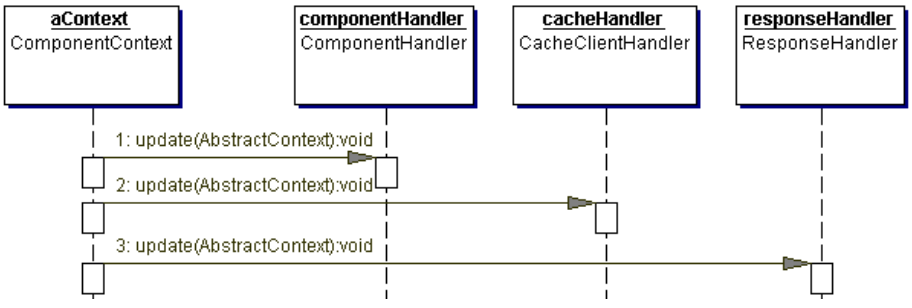

Figura 70 – Cenário de composição de tratadores para a utilização de serviço de *cache*

Tanto o cenário de composição de tratadores para o serviço de *cache* (Figura 70), quanto o cenário de composição de tratadores para um serviço de consulta, reutilizam os tratadores *ComponentHandler* e *ResponseHandler*.

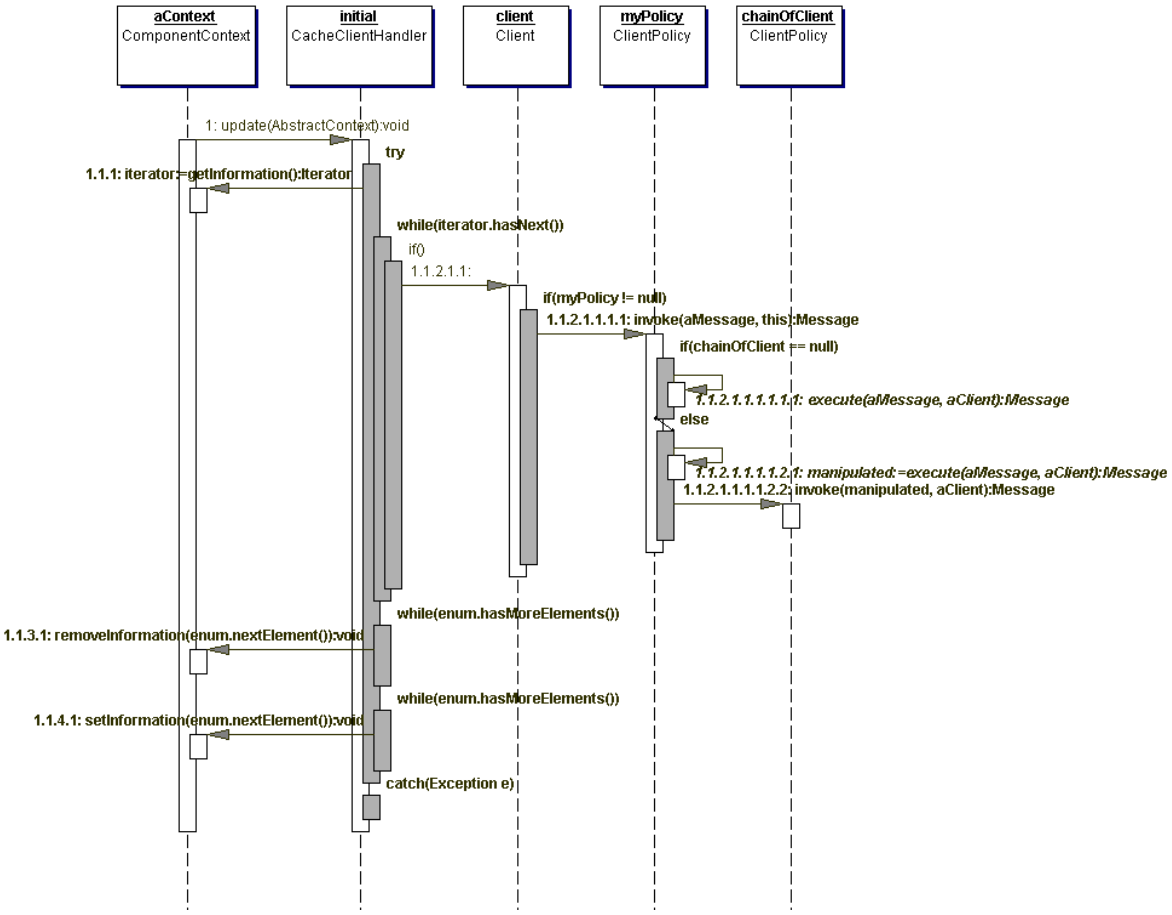

Figura 71 – Exemplo de diagrama de seqüência de tratador para serviço de *cache*

Além disto, estas extensões tratam de possíveis exceções que eventualmente podem acontecer no processo de chamada a um serviço de consulta ou de *cache*. Para encapsular informações foi estendido o contexto *ComponentContext*, que basicamente armazena as mensagens recebidas e as informações que estão em processamento. Finalmente, foram implementadas extensões da classe *ClientPolicy* para a conversão semântica de informações dos serviços de acesso ao Google, à estrutura de *cache*, à base de dados do LES e à base de dados local. Na Figura 71 é possível observar o diagrama de seqüência do tratador *CacheClientHandler*.

6.1.6.
Avaliação do Exemplo

Com relação ao desenvolvimento do sistema de busca de publicações acadêmicas, a especificação das regras e das interdependências, necessárias à coordenação das composições por meio de contratos de coordenação, facilitou o processo de elaboração de soluções de reutilização de composições. A possibilidade de lidar com composições sem se preocupar com detalhes de tecnologias também contribuiu com esta tarefa. Além disto, foi possível reutilizar as adaptações necessárias à integração das diferentes tecnologias de componentes por meio do *framework* de composição proposto. O catálogo de artefatos disponíveis na arquitetura ACCA contribuiu para limitar e organizar a busca por unidades de construção.

Diversos cenários de evolução podem ser desenvolvidos para comprovar a reutilização verbatim dos artefatos de software, o que contribuiria para identificar a aplicação destas unidades em diversas soluções. As regras de negócio, realizadas via contratos de coordenação, podem ser alteradas, o que a princípio não terá impacto nenhum nos artefatos utilizados. Além disto, novas tecnologias podem ser atendidas e integradas à aplicação, estas alterações gerariam modificações na camada de composição. A realização e a evolução do conceito de composição são feitas por meio de novas implementações para os pontos de flexibilização do *framework* proposto. Ao utilizar novos artefatos de software, seguindo uma tecnologia já implementada, parte das adaptações e conversões necessárias para a integração do novo artefato a solução podem ser reutilizadas.

6.2. Implementação de Padrões de Concorrência

Um outro exemplo que serve para demonstrar o uso da Arquitetura ACCA, em especial a aplicação de contratos na camada coordenação e a simplificação resultante da construção dos componentes, é a implementação de alguns padrões de projeto, classificados como padrões de concorrência em Grand (1998).

6.2.1. Descrição do Problema

A técnica de padrões de projeto (Gamma et al., 1995) é uma abordagem bem difundida para a reutilização de fragmentos de projeto, sendo utilizada para organizar a estrutura de componentes de forma sistemática e simples. O uso desta técnica implica em componentes organizados de acordo com um protocolo estrito. Um protocolo é considerado estrito quando é imposto por meio de subclasses de uma classe específica, por meio da provisão de associações, ou por meio da utilização de métodos específicos pré-definidos.

Em alguns casos, a utilização da técnica de padrões de projeto pode ser questionável, pois as formas apresentadas para a organização de interações entre componentes podem não facilitar a evolução da solução, especialmente quando esta interação é passível de alteração com a mudança dos requisitos do sistema (Lano et al., 2002).

Todos os sete padrões de concorrência classificados em Grand (1998) foram desenvolvidos uniformemente utilizando a técnica de especificação de contratos descrita na seção 4.2. Porém, nesta dissertação somente são transcritos os padrões de concorrência: *Scheduler* e *Read/Write Lock* (Lea, 1997).

6.2.2. Interpretação do Problema Segundo ACCA

Os componentes de software para a resolução deste problema são simples classes programadas utilizando Java. Esta decisão foi tomada para que este exemplo destaque as vantagens da separação da camada de coordenação. Isto acaba por simplificar ao extremo a camada composição de software. Uma ilustração para a resolução do problema pode ser vista na Figura 72.

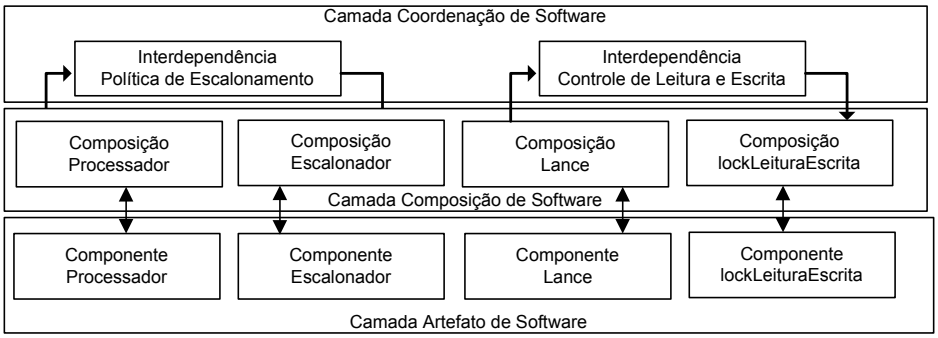

Figura 72 – Ilustração da resolução de padrões de concorrência com ACCA

6.2.3. Scheduler

O padrão *Scheduler* (Lea, 1997) tem como objetivo controlar a ordem em que requisições são escalonadas, encadeando a ordem de execução das requisições em um processador. A partir deste padrão, um processador, ao receber uma requisição, não possui mais controle sobre o momento de sua execução. Para isto, esta requisição deve ser repassada a um escalonador que, ao implementar alguma política de controle de execução, determinará o momento apropriado para a execução da requisição no processador.

6.2.3.1. Especificação do Padrão de Concorrência

Abaixo são ilustradas as interações previstas no padrão *Scheduler* (Grand, 1998) e o seu modelo de coordenação equivalente (Figura 73).

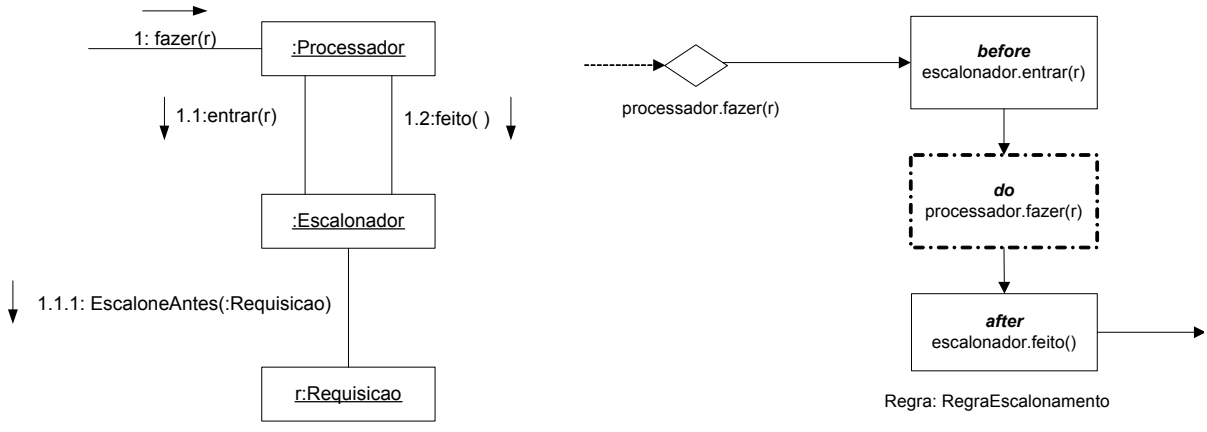

Figura 73 – Interações do padrão *Scheduler* e modelo de coordenação correspondente

Um exemplo de implementação do componente Processador (Grand, 1998) pode ser visto abaixo (Figura 74). É possível observar que as interações previstas no padrão são aplicadas diretamente no componente. Neste caso, é possível compreender que haverá uma modificação no componente de software caso haja

alguma alteração neste protocolo, ou ainda caso haja a troca do escalonador responsável por controlar este processo. Estas alterações são intrusivas, pois o artefato deveria ser uma unidade estável. Este artefato somente deveria sofrer alterações caso a funcionalidade, ao qual ele se presta, sofresse alguma evolução.

```
class Processador {
    private Escalonador escalonador = new Escalonador ();
    public void fazer(Requisicao r) {
        try {
            escalonador.entrar(r);
            try { ...//Processamento }
            finally { escalonador.feito(); }
        } catch (InterruptedException e) { ... } //try
    } //
} // class Processador
```

Figura 74 – Componente Processador implementado de forma *ad hoc*

6.2.3.2. Realização do Estudo em ACCA

Abaixo estão transcritos os códigos correspondentes ao contrato de coordenação (Figura 75), representando as interações previstas no padrão de concorrência, e ao novo componente Processador (Figura 76), sem apresentar características impostas pelo padrão de concorrência.

```
contract ContratoEscalonador
    participants
        processador:Processador;
        escalonador:Escalonador;
    coordination
    RegraEscalonamento:
       when *->> processador.fazer(r)
             before{
                try {
                    escalonador.entrar(r);
                }catch (InterruptedException e){ ... }
             }
             after {
                escalonador.feito();
             };
end contract
```

Figura 75 – Conceitos de coordenação do padrão *Scheduler*

```
public class Processador {
    public void fazer (Requisicao r)
    { ... //Processamento    }
} // class Processador
```

Figura 76 – Artefato de software correspondente ao componente Processador

É possível observar com este exemplo que qualquer alteração relativa ao protocolo imposto pelo padrão de concorrência está restrita ao contrato de coordenação, preservando assim o artefato de alterações desnecessárias.

6.2.4.
Read/Write Lock

O padrão *Read/Write Lock* (Lea, 1997) permite que acessos de leitura concorrente a um objeto ocorram, porém requer exclusividade de acesso para operações de escrita. Este requisito pode ser atendido por meio da coordenação de chamadas concorrentes a métodos de leitura e escrita de informações em um objeto. Neste caso, chamadas de escrita e suas alterações não podem interferir com chamadas deste mesmo tipo, ou com outras chamadas de leitura.

6.2.4.1.
Especificação do Padrão de Concorrência

Abaixo são ilustradas as interações previstas no padrão *Read/Write Lock* (Grand, 1998; Figura 77) e o seu modelo de coordenação equivalente (Figura 78).

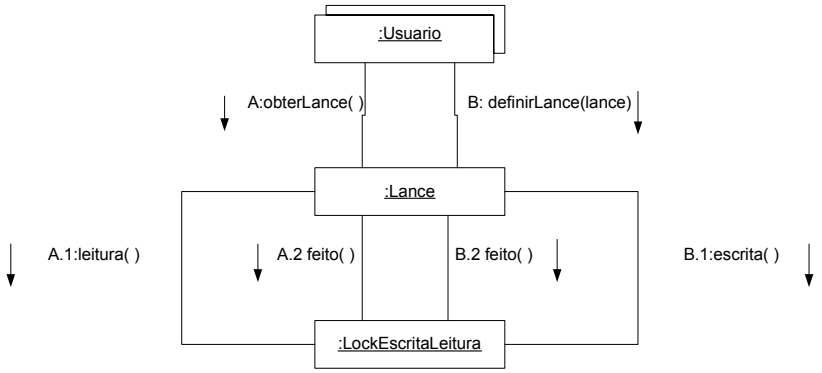

Figura 77 – Interações do padrão *Read/Write Lock*

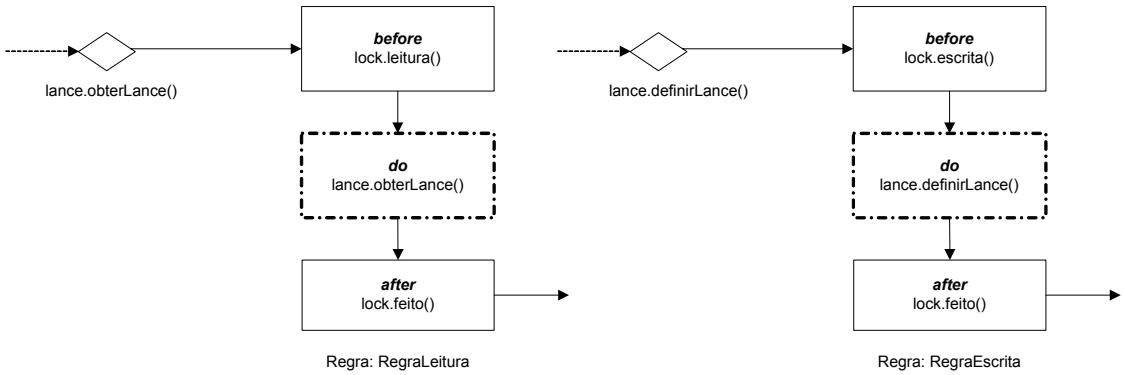

Figura 78 – Modelo de coordenação do padrão *Read/Write Lock*

Um exemplo de implementação do componente Lance (Grand, 1998), representando o componente que é controlado, pode ser visto abaixo (Figura 79). O mesmo fato observado no exemplo anterior pode ser confirmado agora, isto é, o protocolo imposto pelo padrão de concorrência se encontra permeando o código do componente de software.

```
public class Lance {
   private int lance = 0;
   private LockEscritaLeitura lock = new LockEscritaLeitura();
   public int obterLance () throws InterruptedException {
        lock.leitura ();
        int lance = this.lance;
        lock.feito();
        return lance;} // getLance()
   public void definirLance (int lance) throws
InterruptedException {
        lock.escrita();
        if (lance > this.lance) {
            this.lance = lance;
        } // if
        lock.feito();
    }
} // class Lance
```

Figura 79 – Componente Lance implementado de forma *ad hoc*

6.2.5. Realização do Estudo em ACCA

Abaixo estão transcritos os códigos correspondentes ao contrato de coordenação (Figura 80) e ao novo componente Lance (Figura 81), já transcrito sem apresentar características impostas pelo padrão de concorrência.

```
contract ContratoLockLeituraEscrita
    participants
        lance:Lance;
        lock:LockEscritaLeitura;
    coordination
    RegraLeitura:
       when *->> lance.obterLance()
         before {
            try {   lock.leitura();
            } catch (InterruptedException e) { ... } }
         after {
            lock.feito(); };
    RegraEscrita:
      when *->> lance.definirLance(lance)
         before {
            try {   lock.escrita();
            }catch (InterruptedException e) { ... } }
         after{
            lock.feito ();  };
end contract
```

Figura 80 – Conceitos de coordenação do padrão *Read/Write Lock*

```
public class Lance {
   private int lance = 0;
    public int obterLance() {
        return lance;   }
    public void definirLance(int lance){
        if (lance > this.lance)
           { this.lance = lance; } // if   }
 }
```

Figura 81 – Artefato de software corresponde ao componente Lance

O código presente na classe Lance está restrito às funcionalidades previstas para este componente. Qualquer evolução ou alteração na forma de controle de

acesso a métodos desta classe deve ser imposta via contratos de coordenação, preservando a unidade de software, tornando-a mais estável a alterações.

6.2.6. Avaliação do Exemplo

Como foi possível observar nos exemplos de padrões de concorrência, o tratamento da evolução do conceito de coordenação ocorre por meio da utilização de linguagens de alto nível. Isto facilitou a prescrição do conceito de coordenação, contribuindo para um tratamento mais adequado de possíveis evoluções nas interdependências entre as unidades de uma solução.

Além disto, com os dois exemplos apresentados foi possível concluir que as regras impostas pelo padrão de concorrência foram implementadas de forma não intrusiva. Contribuiu para isto a estruturação do conceito coordenação em uma camada isolada e a utilização de contratos prescritivos de coordenação.

O componente resultante presente na camada Artefato de Software, não apresentou características influenciadas pelas interações necessárias a execução do padrão de concorrência. Com isto, foi possível isolar o componente ou artefato de software das regras necessárias para a implementação do protocolo de coordenação imposto pelo padrão. Caso estas regras sejam passíveis de evolução, a evolução será prescrita em um contrato representando o modelo de execução da camada de coordenação de ACCA, o que a princípio não terá nenhuma conseqüência direta sobre o componente utilizado.

Outro ponto interessante, porém não explorado no exemplo, é a total independência da variedade de tecnologia utilizada, isto é, a camada de composição abstrairia todos os detalhes e minúcias necessárias à integração semântica e sintática dos componentes de software. No caso do padrão de concorrência *Scheduler*, seria possível associar um escalonador, implementado utilizando CORBA, a um processador implementado utilizando a tecnologia de *Web Services*. Com relação ao padrão *Read / Write Lock*, a estrutura de controle *LockLeituraEscrita* seria implementada utilizando uma proposta de arquitetura de agentes de software e o Lance poderia ser implementado via Java/EJB.