4
Realização de ACCA

Resumo

Neste capítulo é apresentada uma proposta para a realização das camadas arquiteturais, propostas em ACCA. Para auxiliar o entendimento deste capítulo é apresentado o problema de marcação de compromissos. Com relação à realização das camadas, é descrita a proposta de contratos de coordenação apresentada em Lano et al. (2002) para a realização da camada de coordenação. Em seguida, é apresentado um framework desenvolvido especialmente para a realização da camada de composição. Finalmente, é recomendado um conjunto de ferramentas de apoio ao armazenamento e à catalogação de artefatos de software, que são um exemplo da realização das abstrações e dos serviços necessários à camada Artefatos de Software de ACCA.

4.1.
O Problema Exemplo

A marcação de compromissos e reuniões é o exemplo utilizado neste capítulo para auxiliar a entender como que a realização dos conceitos de ACCA pode contribuir para o desenvolvimento de soluções baseadas em componentes. Este exemplo foi inspirado no estudo de caso proposto em (Ferreira et al., 2003).

4.1.1.
Marcação de Compromissos

Atualmente, a integração de diferentes sistemas de marcação de compromissos ocorre de maneira limitada, isto é, não existe uma forma integrada de agendar compromissos com participantes que utilizam diferentes sistemas para esta finalidade.

Suponha que todos os serviços de marcação de compromissos existentes estão acessíveis via Internet e que cada sistema dispõe de pelo menos três operações básicas: marcar compromisso, desmarcar compromisso e consultar disponibilidade. Cada serviço é disponibilizado por meio de um provedor de

serviços. Este provedor recebe requisições no formato de algum protocolo de acesso (por exemplo, SOAP), utilizando algum protocolo de transporte (por exemplo, SMTP ou HTTP). Uma das características intrínsecas ao problema é uma possível indisponibilidade dos serviços durante o processo de tomada de decisão. Boa parte dos serviços de marcação de compromissos está disponibilizada em PCs, telefones celulares ou PDAs que podem ser desligados ou desconectados da rede periodicamente.

É importante observar que uma maior variedade de sistemas de agenda, implica em uma maior quantidade de detalhes que precisam ser considerados na composição de artefatos, necessária ao processo integrado de marcação de compromissos.

4.1.2. Marcação de Compromissos: Cenário de Uso

Neste cenário de uso, deseja-se marcar um encontro entre um grupo de participantes (Figura 19). Porém, para que este encontro ocorra, cada participante envolvido no compromisso impõe um conjunto de dependências ou restrições (Figura 19, Figura 20). Todas as dependências podem ser descritas utilizando-se regras (Figura 20) que restringem as alternativas possíveis para que um participante esteja presente em um compromisso.

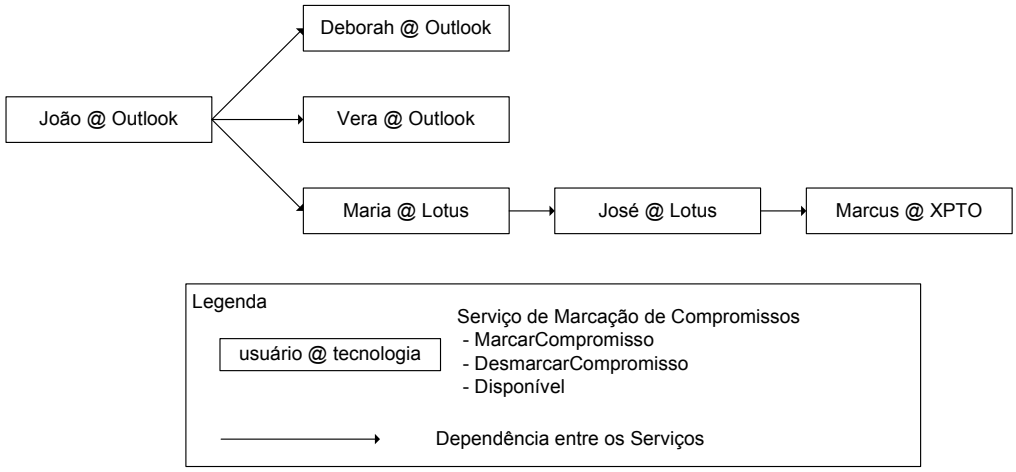

Figura 19 – Ilustração de cenário de serviços de marcação de compromissos

Deseja-se marcar um compromisso cuja previsão de duração é de duas horas, na semana de 22 de agosto de 2003. No cenário elaborado (Figura 19, Figura 20), João é diretor executivo da empresa A e utiliza o Microsoft Outlook® como o seu serviço de agenda. Agora imagine uma situação em que João pretenda agendar um compromisso com Deborah e Vera, que trabalham na mesma empresa

A, e com Maria da empresa B (Figura 19). A empresa B adota o IBM Lotus Notes® como serviço de agenda.

Analisando com mais atenção o conjunto de dependências existentes, é possível observar que Maria, da empresa B, requer que o seu companheiro de trabalho José esteja presente na reunião. Por sua vez, José requer a presença do consultor externo Marcus, que utiliza o cliente XPTO para marcação de compromissos.

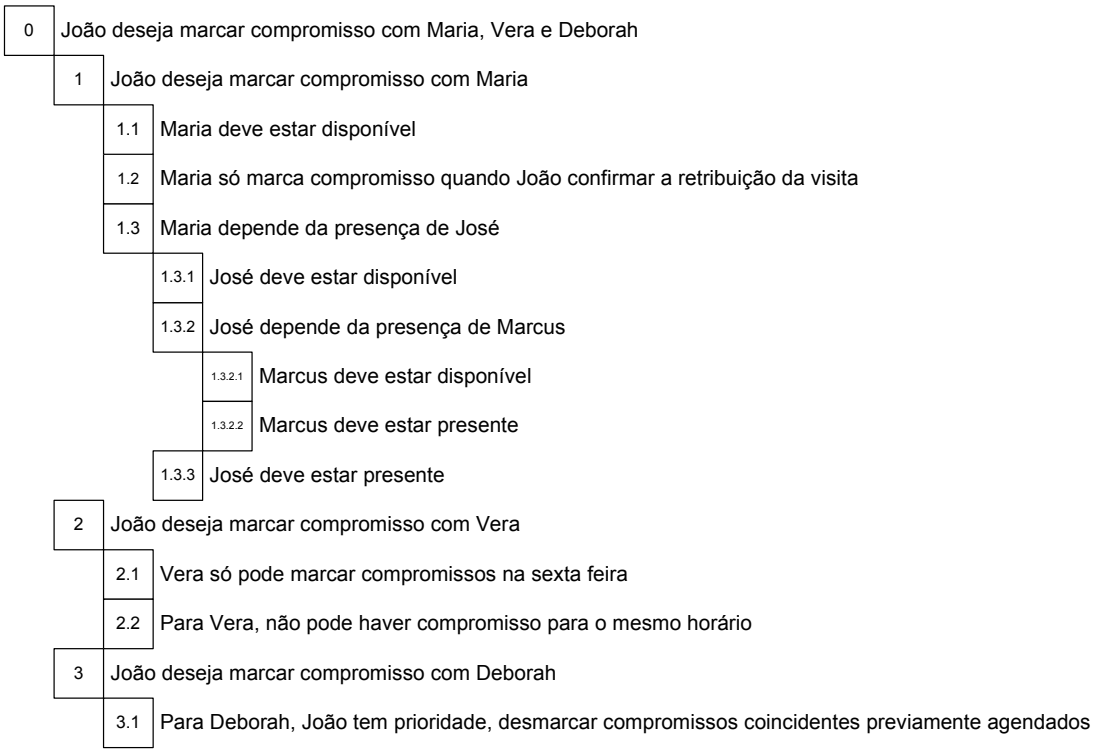

Figura 20 – Regras para o cenário de uso de marcação de compromissos

4.2. Contratos de Coordenação

Para ilustrar a realização da camada de coordenação de ACCA, neste trabalho é utilizada a proposta apresentada em Lano et al. (2002), chamada de contratos de coordenação. Esta abordagem foi escolhida, pois apresenta uma ferramenta de apoio ao desenvolvimento de soluções (ATX Software, 2002; Gouveia et al., 2001) utilizada para auxiliar o processo de geração de código. Além desta abordagem, existem outras propostas alternativas que poderiam ser utilizadas com pequenas adaptações para esta mesma atividade (Hollingsworth, 1995; IBM BPEL, 2002; Workflow, 2003).

A utilização de contratos de coordenação é apresentada em Lano et al. (2002) como forma de organizar aplicações em termos de regras de negócio e outras invariantes necessárias para a integração e a restrição do comportamento conjunto dos componentes aos quais o contrato se aplica.

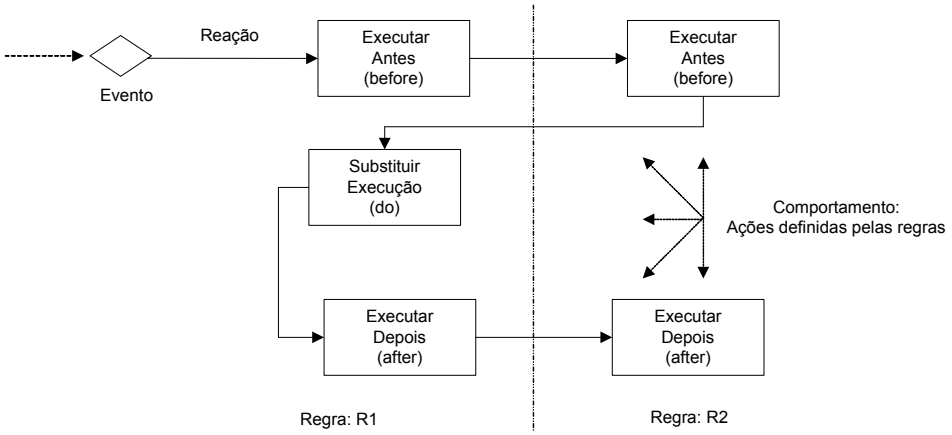

Figura 21 – Modelo de coordenação utilizado (Lano et al., 2002)

Neste modelo (Figura 21), baseado nos conceitos de superposição (Katz, 1993), chamadas explícitas a um método correspondem a um evento e são desacopladas da execução do serviço, permitindo assim que as regras definidas no contrato interceptem a chamada. Após esta interceptação, é possível encadear ou sobrepor o comportamento necessário para gerar a solução desejada. Este mecanismo é especificado por meio de regras do tipo evento → reação. Este conjunto de regras pode incluir ações antes ou depois da execução de determinado serviço, além de opcionalmente substituir o comportamento a ser executado.

Abaixo são descritos os campos utilizados na especificação de um contrato para a coordenação de componentes de software.

```
contract NomeContrato
      participants ...
      attributes ...
      operations ...
coordination
      Nome Regra:
            when *->> evento && (condições)
                  before { ... }
                  do { ... }
                  after { ... }
end contract
```

Figura 22 – Contrato de coordenação: sintaxe

O cabeçalho de um contrato define o nome do contrato e os participantes envolvidos. Em ACCA, estes participantes correspondem às composições sujeitas às regras de coordenação. Além disto, em um cabeçalho é possível definir novos

métodos que podem ser utilizados na execução do contrato, e novos atributos passíveis de manipulação que representam o estado do processo de coordenação.

No restante do corpo do contrato são especificadas as regras que relacionam eventos e condições a tratamentos específicos. A cláusula *when* identifica justamente as condições e os eventos, e associa a esta restrição procedimentos que serão executados antes ou depois de sua ocorrência, especificados pelas diretivas *before* e *after,* respectivamente. Nesta definição de regras ainda é possível substituir a ocorrência de um evento, sobrepondo o seu comportamento padrão por um procedimento determinado pela diretiva *do*. Estes procedimentos e comportamento são prescritos utilizando Java como linguagem de programação.

4.2.1. A Camada de Coordenação de ACCA

No caso da técnica aplicada para a realização da camada de Coordenação de ACCA (Figura 23), existe uma aplicação, chamada *Coordination Development Environment* (ATX Software, 2002), responsável pela geração de código (Gouveia et al., 2001).

Em tempo de construção, esta ferramenta é utilizada para converter as especificações contidas em um contrato em um conjunto de códigos-fonte correspondentes à implementação do padrão de coordenação proposto em (Andrade et al., 2000).

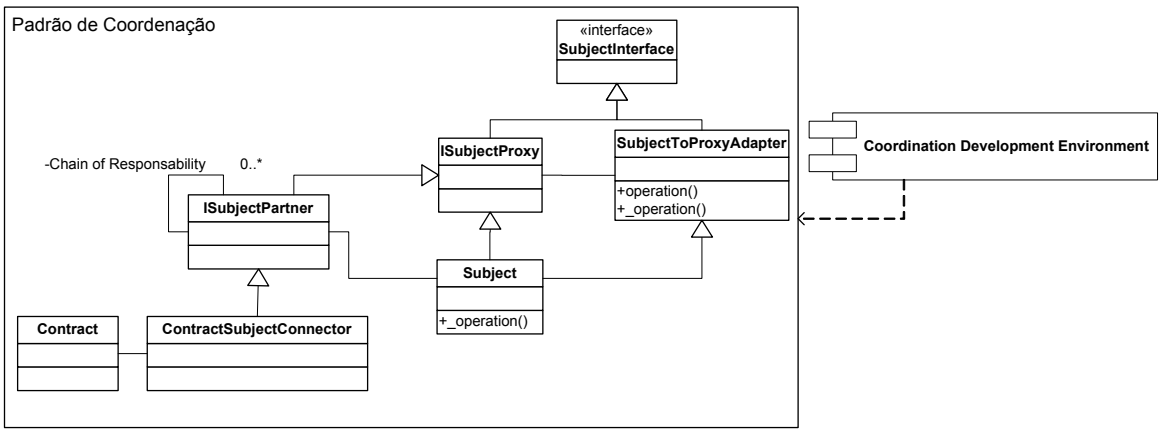

Figura 23 – Realização da camada de coordenação

A partir de um contrato e de componentes Java é gerada toda a estrutura necessária ao processo de coordenação. Neste trabalho, estes componentes Java são restritos a interfaces criadas para a interação com construtos da camada de

composição de ACCA. Portanto, participam do processo de coordenação somente instâncias de implementações desta interface.

Na Figura 23, o padrão de coordenação apresentado (Andrade et al., 2000) correspondente ao projeto da proposta de contratos de coordenação (Lano et al., 2002). Nesta proposta é possível identificar o padrão de projeto *Chain of Responsability* (Gamma et al., 1995), que é a estrutura responsável por implementar o encadeamento e a seqüência de ações necessárias para a realização do modelo de coordenação proposto (Figura 21).

4.2.2. O Problema Exemplo: Realização da Camada de Coordenação

Em Ferreira et al. (2003), um algoritmo baseado no *majority consensus* (Thomas, 1979) foi utilizado para agendar compromissos em grupo e obter consenso em possíveis situações de conflito.

Abaixo são transcritos alguns contratos de coordenação que correspondem à extensão do algoritmo utilizado. Esta extensão corresponde a implementação das regras descritas na Figura 20. Antes de cada contrato, para facilitar o seu entendimento, as regras são detalhadas em conjunto com um modelo de coordenação equivalente.

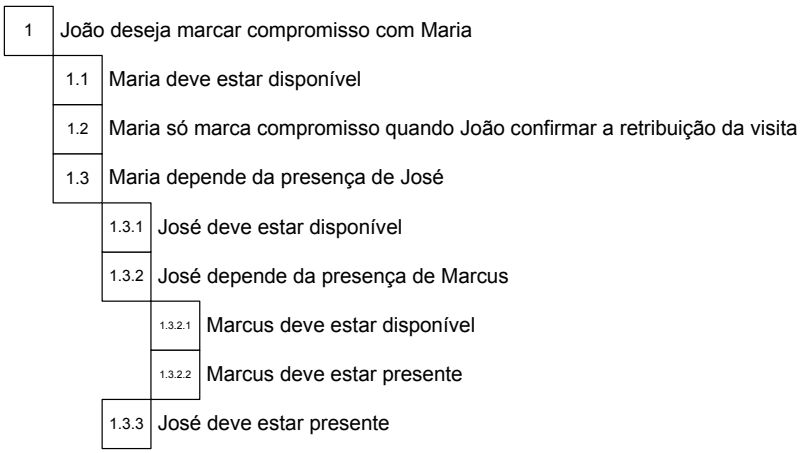

Figura 24 – Regras impostas para compromissos entre João, Maria, José e Marcus

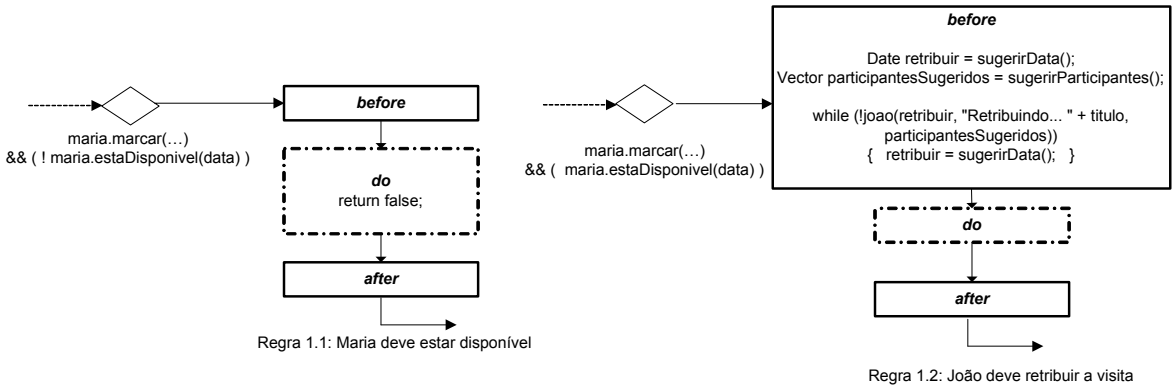

Figura 25 – Restrições 1.1 e 1.2 de Maria

```
contract JoaoMaria
  participants
    joao:Agenda;
    maria:Agenda;
  operations
   public Date sugerirData(){
     /* método que sugere data para marcar retribuição a visita */
   }
   public Vector sugerirParticipantes(){
     /* método que sugere participantes de retribuição a visita
      *      maria e joao serão os participantes sugeridos       */
   }
  coordination
    MariaDeveEstarDisponivel:
    when *->> maria.marcar(data, titulo, participantes)
              && (!maria.estaDisponivel(data))
                do{
                      return false;
                 };
    JoaoDeveRetribuirVisitaMaria:
    when *->> maria.marcar(data, titulo, participantes)
             && (maria.estaDisponivel(data))
     before{
              Date retribuir = sugerirData();
              Vector participantesSugeridos =
                          sugerirParticipantes();
               while (!joao(retribuir, "Retribuindo... " +
                      titulo, participantesSugeridos)) {
                   retribuir = sugerirData();
               }
     };
end contract //JoaoMaria
```

Figura 26 – Contrato com as regras de Maria para marcar compromissos com João

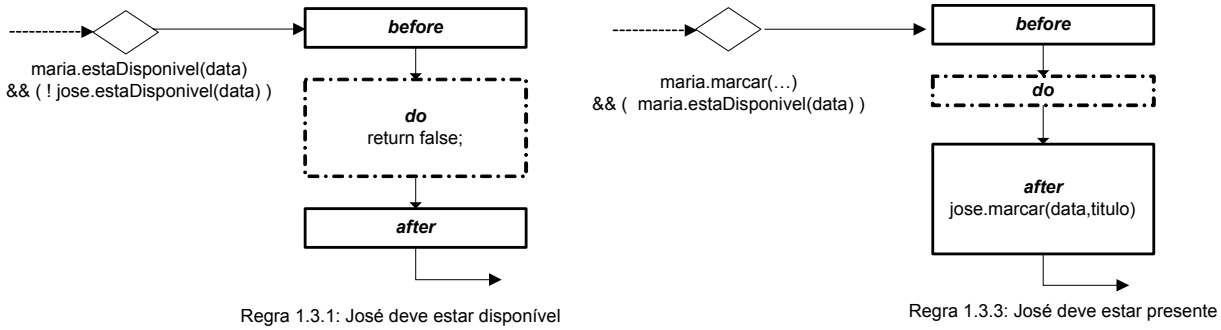

Figura 27 – Restrições 1.3.1 e 1.3.3 de Maria e José

```
contract MariaJose
    participants
        maria:Agenda;
        jose:Agenda;
```

```
    coordination
      JoseDeveEstarDisponivel:
         when *->> maria.estaDisponivel(data)
                && (!jose.estaDisponivel(data))
                    do{
                        return false;
                    };
      JoseDeveEstarPresente:
         when *->> maria.marcar(data, titulo, participantes)
                && (maria.estaDisponivel(data))
                   after{
                    jose.marcar(data,titulo,participantes);
                   };
end contract //MariaJose
```

Figura 28 – Contrato com as regras de José para marcar compromissos com Maria

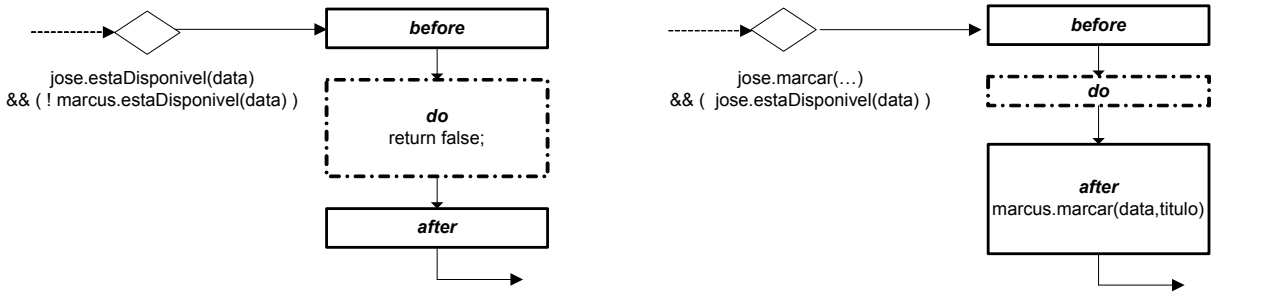

Figura 29 – Restrições 1.3.2.1 e 1.3.2.2 de Marcus e José

Como descrito na Figura 29 e Figura 27, o contrato que expressa as regras de Marcus para marcar compromissos com José é equivalente ao contrato apresentado na Figura 28 e portanto não será transcrito novamente.

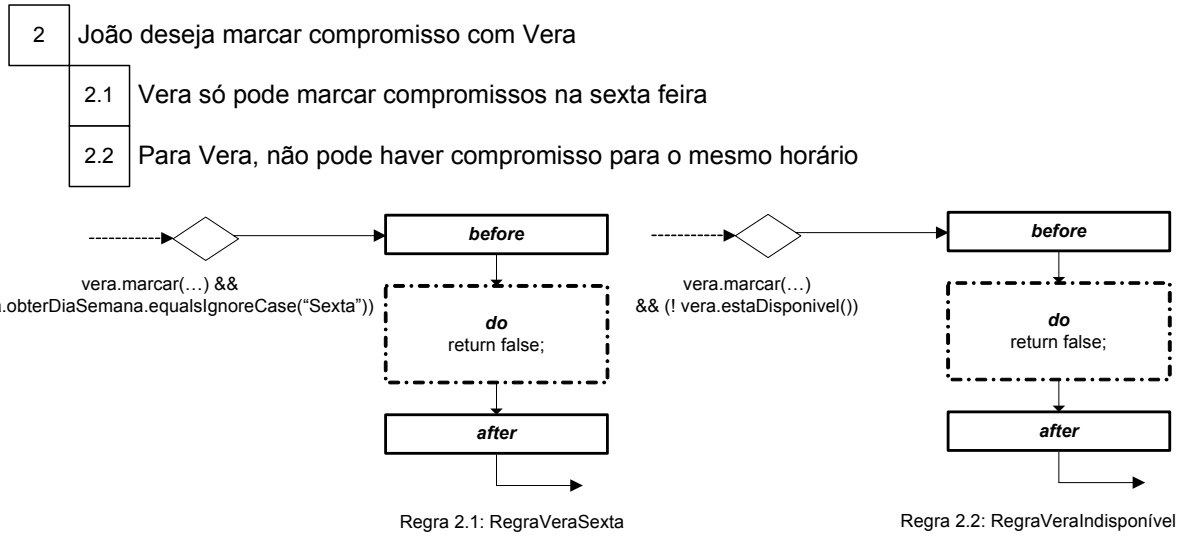

Figura 30 – Dependências para compromissos entre Vera e João

```
contract JoaoVera
 participants
   joao:Agenda;
   vera:Agenda;
 coordination
  RestricoesVera:
   when *->> vera.marcar(data, titulo, participantes)
   && ((!vera.estaDisponivel(data))
      || (!data.obterDiaSemana().equalsIgnoreCase("Sexta")))
            do{
                   return false;
            };
end contract //JoaoVera
```

Figura 31 – Contrato com as regras de Vera para marcar compromissos com João

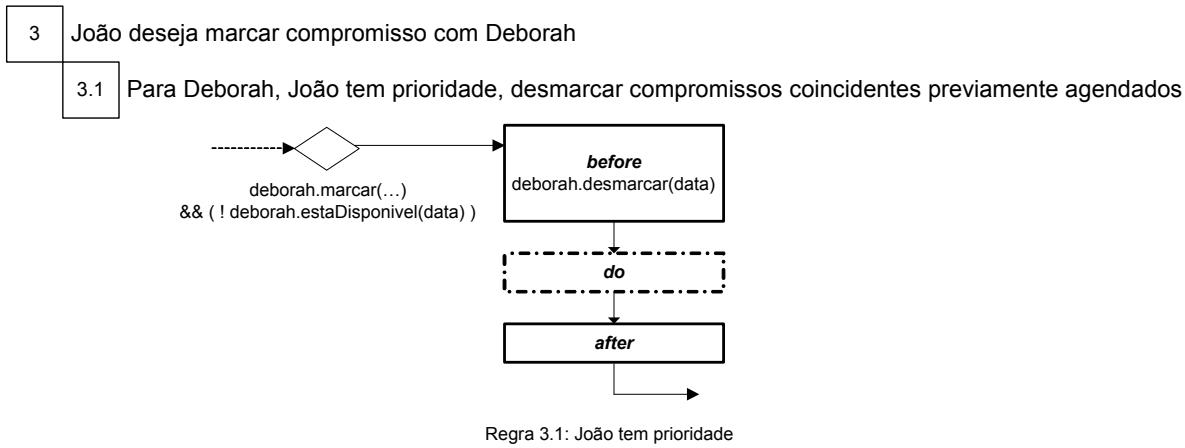

Figura 32 – Dependências para a marcação de compromisso entre Deborah e João

```
contract JoaoDeborah
    participants
      joao:Agenda;
      deborah:Agenda;
    coordination
      RestricoesDeborah:
      when *->> deborah.marcar(data, titulo, participantes)
                && (!deborah.estaDisponivel(data))
          before{
                  deborah.desmarcar(data);
          };
end contract //JoaoDeborah
```

Figura 33 – Contratos com as regras de Deborah para marcar compromissos com João

4.3. Um *Framework* para a Composição de Artefatos de Software

A máquina de composição proposta nesta dissertação procura oferecer uma solução flexível capaz de permitir a invocação de qualquer tipo de componente, independente da tecnologia utilizada para sua implementação. Isto acaba permitindo que diferentes tecnologias convivam como partes de um único sistema.

Além da gerência de protocolos de comunicação e da invocação dos componentes, é oferecida uma infra-estrutura básica para a reunião de artefatos de software em construções mais complexas.

A abordagem de *frameworks* orientados a objeto (Johnson, 1997; Fayad,1999) foi escolhida por permitir que o tratamento de protocolos e as peculiaridades de cada tecnologia sejam transformados em pontos de flexibilização, passíveis de extensão. Além disto, com o uso da técnica de *framework* é possível combinar diferentes modelos de composição por meio da implementação dos diferentes pontos de flexibilização disponíveis. Outra provável vantagem da utilização desta abordagem é a possibilidade de reutilizar adaptações, necessárias ao convívio entre tecnologias de componentes, em diferentes instâncias do *framework*.

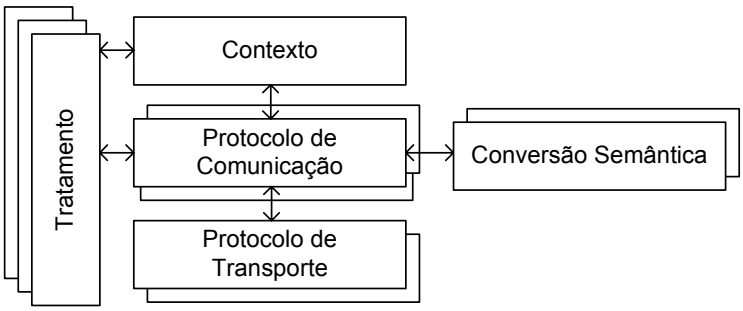

Figura 34 – Pontos de flexibilização do *framework* de composição

Ao todo, são cinco os principais pontos de flexibilização do *framework* (Figura 34): o protocolo de transporte (por exemplo, HTTP e SMTP), o protocolo de comunicação (por exemplo, SOAP), a conversão utilizada para promover o entendimento semântico, a forma de controle do acesso a informações em um contexto e a forma de tratamento interno das informações.

Estes pontos de flexibilização permitem que o *framework* seja instanciado para diferentes tecnologias de componentes sem que o núcleo de composição seja reconstruído. O núcleo de composição implementado corresponde a um fluxo padrão de distribuição interna de mensagens. O núcleo também é a parte responsável por organizar todas as políticas ou as restrições necessárias à ocorrência de composições. Além disto, ele fornece uma estrutura básica para o compartilhamento de informações e de serviços ao grupo de composições existente.

Uma conseqüência indireta da instanciação dos pontos de flexibilização propostos é o modelo de chamada dos componentes. Uma chamada a um

componente pode ser síncrona ou assíncrona. Para que seja possível a convivência destes dois modelos, o método de envio de mensagens é desacoplado do método de recebimento. Com esta abordagem, é possível emular o sincronismo por meio da seqüência de envio de uma requisição e da consulta contínua por uma resposta. No caso de componentes distribuídos, o sincronismo sempre foi uma artimanha de execução implementada para simular uma comunicação contínua entre as partes (Frolund, 1996).

4.3.1. Projeto do *Framework* de Composição

A Figura 35 apresenta uma visão geral do *design* do *framework* com todas as entidades necessárias para a composição, o monitoramento e a invocação de componentes.

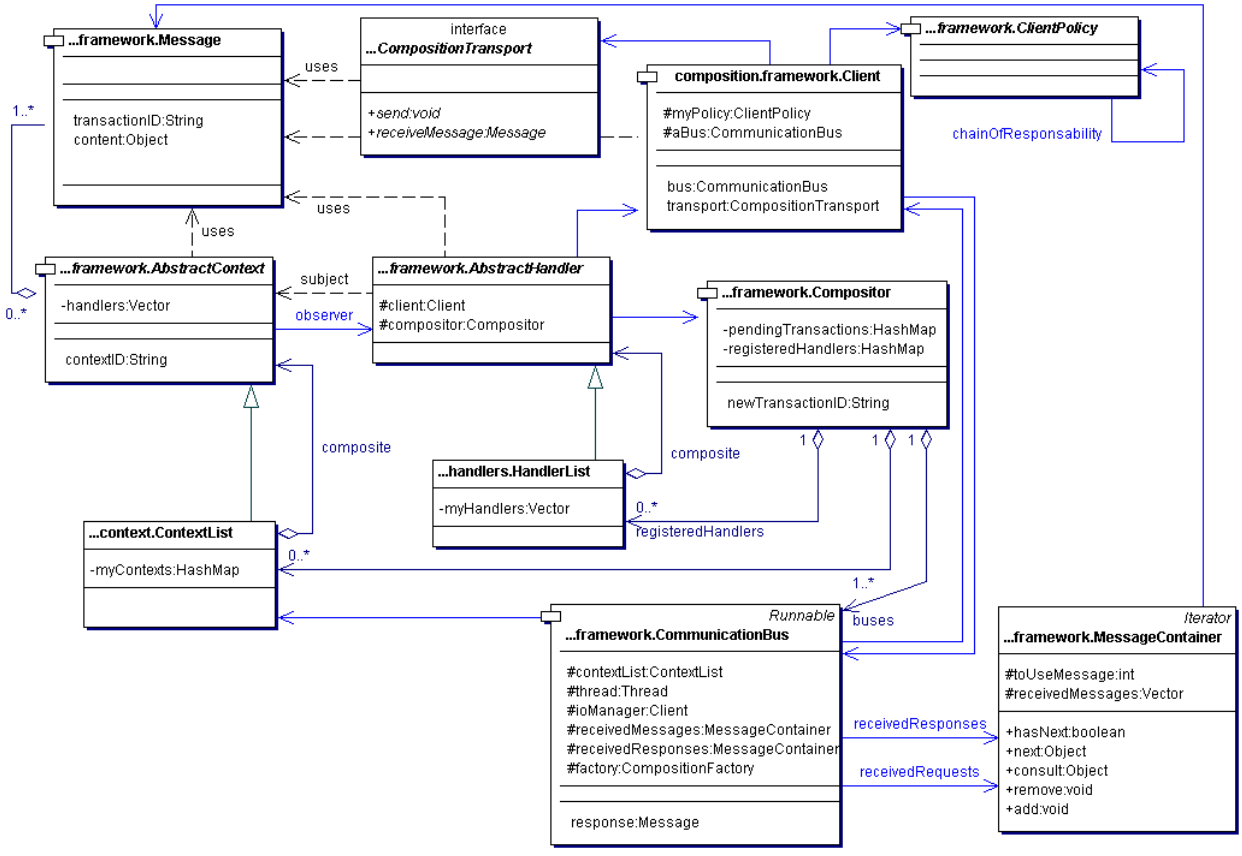

Figura 35 – Visão geral do *framework*

O núcleo deste *framework* oferece uma estrutura básica de composição, implementada na estrutura *Context-Handler* (Figura 36), por meio do padrão de projeto *Observer* (Gamma et al., 1995). Esta estrutura é responsável pela manutenção de estados e do fluxo de execução das ações. Por meio das classes

AbstractContext e *AbstractHandler*, é possível manter o estado de execução dos processos e associar tratadores aos estímulos esperados, respectivamente. Com esta abordagem, todos os componentes residentes em um contexto podem potencialmente interceptar e manipular cada chamada que ultrapasse a fronteira fictícia definida pelo contexto. Cada componente pode contribuir com as informações essenciais para o seu contexto ou ainda, usufruir os serviços oferecidos pelo *framework* ou pelas demais unidades presentes em uma composição.

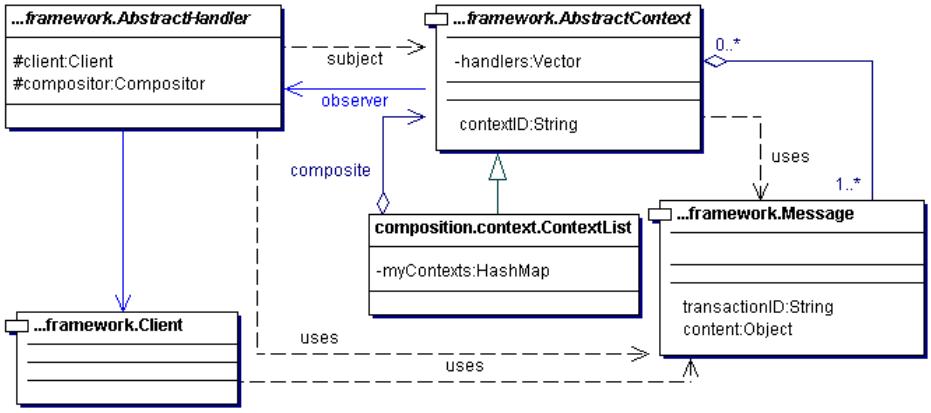

Figura 36 – *Framework* foco na classe *Context*

A classe *AbstractContext* (Figura 36) é um *Subject* do padrão Observer (Gamma et al., 1995), implementado pela estrutura *Context-Handler*. Sua função principal é encapsular o estado do tratamento de requisições externas e manipulações internas comuns em um processo de composição. Um contexto ou uma lista de contextos recebe mensagens (instâncias de *Message*) por meio de estímulos transmitidos por um barramento, uma instância da classe *CommunicationBus.* Além disto, a influência ou propagação de uma informação pode ser organizada hierarquicamente em um contexto por meio do padrão *Composite* (Gamma et al., 1995).

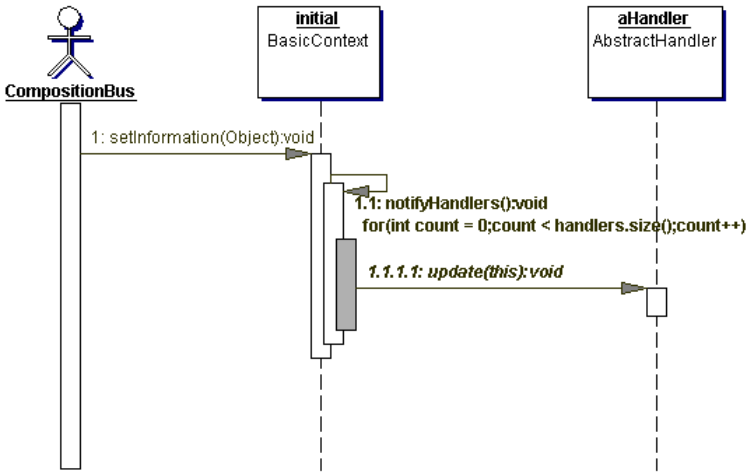

Figura 37 – A seqüência para tratamento de um estímulo a um contexto

Ao receber um estímulo, um contexto pode modificar o seu estado interno. Caso haja esta modificação, uma notificação é enviada a tratadores, que são os possíveis interessados nesta informação (Figura 37). Quando necessário, tratadores correspondentes às instâncias de *Handlers* ou composições de *Handlers* podem chamar *Clients* de componentes responsáveis pela execução de um serviço oferecido por um artefato de software.

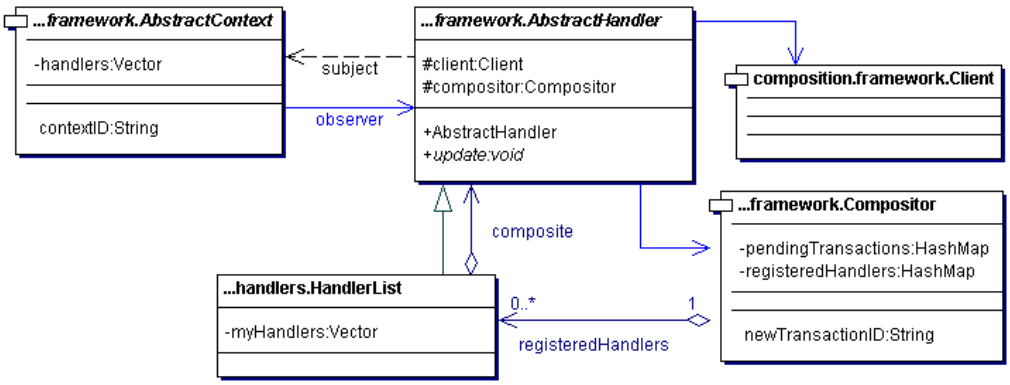

Figura 38 – *Framework* foco na classe *Handler*

Um *AbstractHandler* (Figura 38) é um *Observer* (Gamma et al., 1995) da estrutura *Context-Handler*. Um *Handler* é chamado no momento em que ocorre uma alteração relevante em algum contexto ao qual ele está associado (Figura 37). Após a análise do contexto alterado, um *Client* pode ser chamado. Esta chamada corresponde à invocação de um artefato de software específico. Além disto, um *Handler* pode ser chamado hierarquicamente a partir do padrão *Composite* (Gamma et al., 1995). Com isto, podem-se associar vários tratadores a um único contexto, e restringir a forma como uma informação é repassada a níveis hierárquicos de tratamento de atualizações.

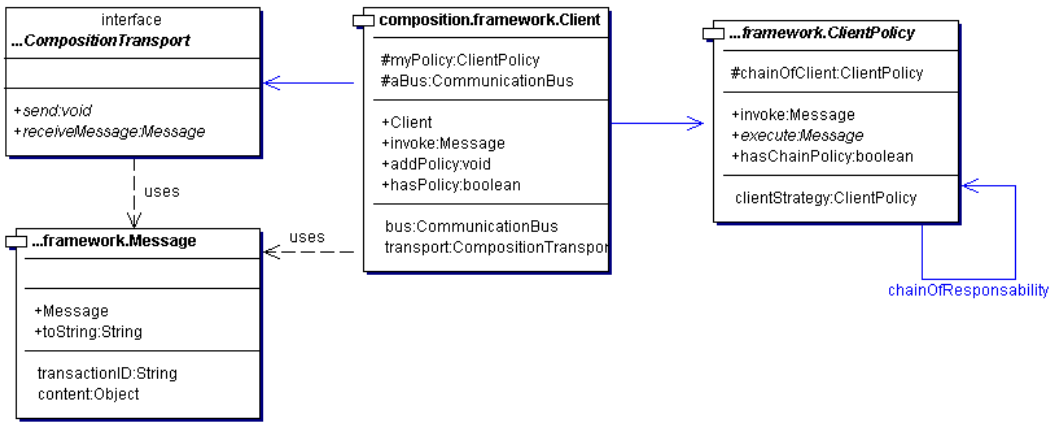

Figura 39 – *Framework* foco nas classes *Client* e *ClientPolicy*

Na abstração criada para esta camada (Figura 14), um *Client* representa um atuador ou sensor e funciona como um *proxy* dinâmico para a invocação de componentes. Este *proxy* pode utilizar uma implementação da interface *CompositionTransport,* que é responsável por lidar com questões específicas do protocolo de transporte. A classe *Client* foi criada para organizar uma cadeia de políticas extensíveis, adequadas para o tratamento de questões de protocolos de comunicação e conversões semânticas, sendo que cada política é implementada por uma extensão da classe *ClientPolicy*. Uma extensão da classe *ClientPolicy* implementa questões relativas a uma tecnologia específica, considerando para isto todas as características e restrições de comunicação e quaisquer outras questões necessárias e relevantes para a utilização integrada do serviço do componente. Por exemplo, esta seqüência pode efetuar conversões semânticas, auxiliando o entendimento de informações trocadas por meio de mensagens. Para esta finalidade, a cadeia de *ClientPolicy* implementa o padrão de projeto *Chain of Responsability* (Gamma et al., 1995).

Um *Compositor* (Figura 40) corresponde a um grupamento lógico de uma série de serviços oferecidos pelo *framework* como catálogo de componentes ou a lista de clientes implementados. Ele é responsável pela execução da máquina de composição como um todo, armazenando todas as referências para os canais de comunicação existentes, contextos criados e seus respectivos tratadores. Além destes serviços e das características descritas, esta classe foi criada para representar a abstração que agrega os diferentes barramentos utilizados para a comunicação interna de mensagens.

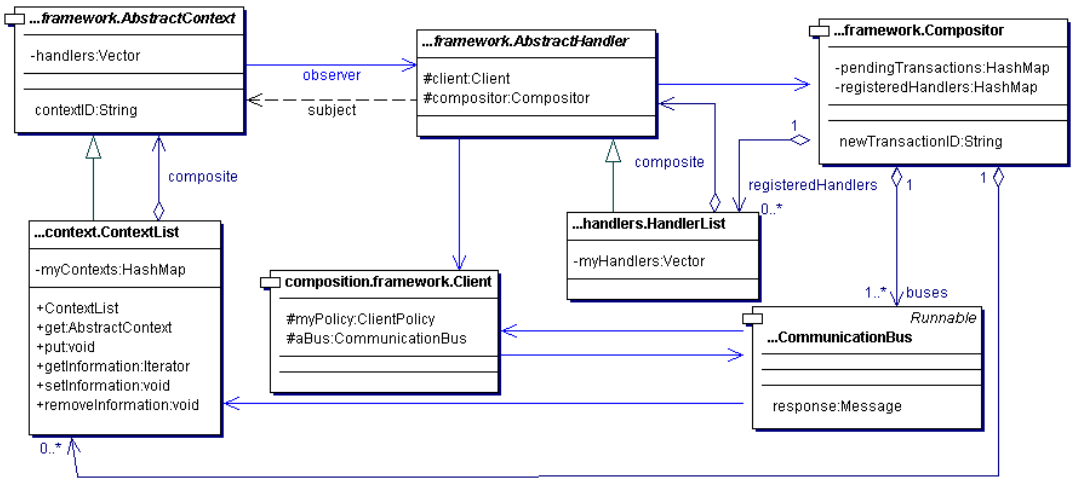

Figura 40 – *Framework* foco na classe Compositor

As mensagens que trafegam em um barramento são obtidas a partir de instâncias da classe *Client* ou por meio de extensões da classe *AbstractHandler*. Nesse caso específico, a classe *Client* pode ser entendida como um gerente de entrada e saída, encapsulando todo o procedimento de comunicação com entidades externas ao *framework* de composição.

A partir de implementações da interface *CompositionTransport,* é possível realizar os diferentes protocolos de transporte utilizados pela estrutura de composição. O conjunto de todas as implementações de *CompositionTransport* realiza a abstração de interface criada para a camada de composição de ACCA. Esta interface é responsável pela interação com os artefatos de software que são passíveis de composição. Exemplos de implementações de *CompositionTransport* incluem realizações de HTTP e SMTP, responsáveis pela implementação de protocolos de transporte síncronos e assíncronos.

A classe *Message* representa uma abstração criada para estruturar as informações trocadas entre os componentes internos do *framework*. Ao receber um estímulo externo, *CompositionTransport* cria uma mensagem, do tipo *Message*, contendo a estrutura e todas as informações relevantes para o processamento de informações. Após fazer esta conversão, a mensagem criada é entregue a um *Client*. Depois de aplicar todas as políticas de comunicação previstas, a mensagem é redirecionada a um barramento específico que distribuirá estas informações entre os componentes internos da solução de integração.

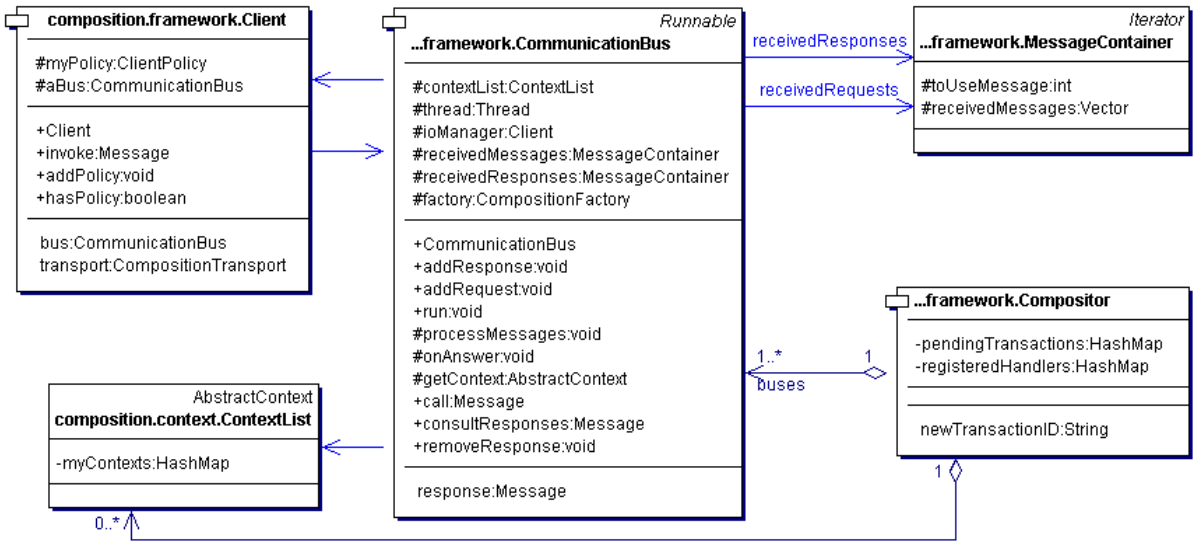

Figura 41 – *Framework* foco na classe *CommunicationBus*

Um barramento (Figura 41), representado pela classe *CommunicationBus,* encapsula o procedimento de anúncio da chegada de informações a um grupo de contextos que lhe são associados. Ele é a entidade responsável pela comunicação entre todos componentes internos da solução. Por exemplo, ele torna viável a comunicação entre o padrão de composição e os estímulos, de entrada e saída do sistema, gerados por sensores e atuadores. Estes sensores ou atuadores tipicamente são implementações que estão de acordo com algum protocolo de comunicação específico. Como já foi visto, os sensores ou atuadores são representados pelas diferentes implementações da classe *Client*.

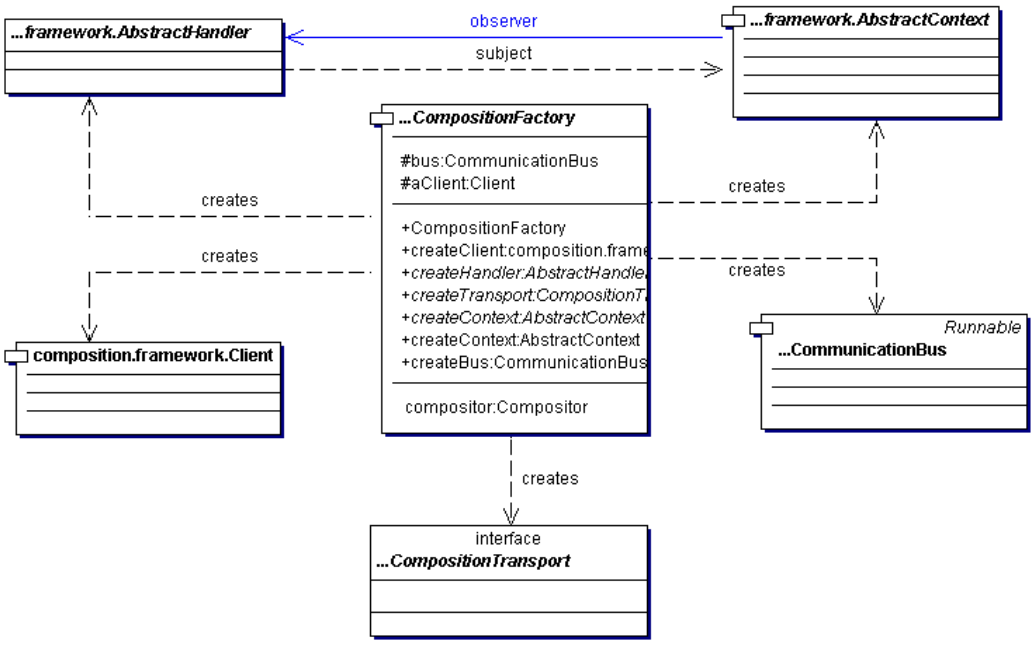

Figura 42 – *Framework* foco na classe *CompositionFactory*

O uso de classes abstratas foi o artifício de *design* que permitiu a criação de uma infra-estrutura comum, reutilizável, e independente das diversas extensões e dos diversos protocolos implementados. Estes princípios tornaram necessária a criação da entidade *CompositionFactory* (Figura 42), que simplifica e organiza a utilização de instâncias do *framework* para composição. Esta entidade, que implementa o padrão de criação *Factory* (Gamma et al., 1995), é responsável por criar os canais de comunicação, os tratadores específicos necessários ao processo de composição, os contextos especializados, necessários para isolar as informações relevantes, e as especializações da classe *Client* ou implementações da interface *CompositionTransport*.

4.3.2. A Camada de Composição de ACCA

Na comparação entre o *framework* proposto e a abstração criada para a camada de composição de ACCA (Figura 14), o barramento corresponde essencialmente a estrutura *CommunicationBus*, sensores e atuadores correspondem a combinação das estruturas *Client* e *ClientPolicy*, a interface corresponde ao conjunto de implementações da interface *CompositionTransport* e, por fim, o núcleo do padrão de composição é implementado principalmente por meio de extensões da estrutura *Context-Handler*.

Uma instância de um *Client* pode ser definida como sensor e atuador de acordo com o modelo de uso do componente. Por exemplo, para modelos síncronos, um *Client* é ao mesmo tempo atuador e sensor. Em modelos assíncronos, *Clients* podem ser sensores ou atuadores, ou ambos, dependendo de decisões de projeto. Padrões de composição são resultantes de instâncias do *framework* de composição. Suas características são determinadas pelo conjunto de extensões feitas.

4.3.3. Reflexão sobre um Cenário de Uso do *Framework*

Por meio das instanciações dos diferentes *hot-spots* propostos, é possível utilizar o *framework* para compor componentes em diversos contextos diferentes. É possível desenvolver soluções que envolvam um conjunto diverso de tecnologias de componentes.

A partir do *framework* proposto, é possível realizar a composição de componentes utilizando diferentes protocolos de transporte. Isto permite que componentes possam ser utilizados independente da maneira como estão disponibilizados.

Outro ponto importante é a independência de protocolo de comunicação com os componentes. Cada tecnologia e cada modelo de execução podem definir um protocolo de comunicação diferente, o que torna necessária a criação de novos invocadores dinâmicos para cada tipo de componente existente. Além disso, por meio do *framework,* é possível flexibilizar também o formato do conteúdo das mensagens processadas, permitindo que qualquer dado, comunicado com algum componente, seja convertido e processado de forma homogênea pelo *framework*.

4.3.4. O Problema Exemplo: Realização da Camada de Composição

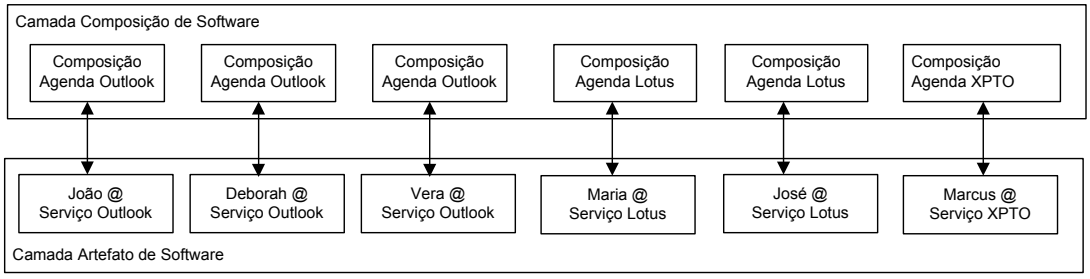

Figura 43 – Ilustração de relacionamento entre as composições e os serviços

Na realização da camada de composição do problema exemplo (Figura 43), cada tecnologia de serviço de agenda é representada por uma única composição. Portanto existem três composições básicas: a composição Outlook®, a composição Lotus Notes® e a composição XPTO. Para usuários de um mesmo tipo de tecnologia, são criadas instâncias de uma mesma composição. Por exemplo, João, Vera e Deborah utilizam a mesma composição do serviço do Outlook®, porém com diferentes instâncias para cada um.

Com relação ao *framework*, para realizar a camada de Composição neste problema exemplo (Figura 44), foi necessário implementar a interface *CompositionTransport* para realizar os protocolos de transporte HTTP e SMTP, permitindo que os estímulos externos a composição fossem recebidos pela aplicação independente do protocolo utilizado. No caso do exemplo apresentado, o uso do transporte assíncrono é recomendável a usuários que mantém as máquinas desligadas durante boa parte do dia. Neste caso, o uso de transporte

HTTP pode inviabilizar a marcação de compromissos enquanto a máquina estiver desligada. Por exemplo, as chamadas via SMTP podem ficar armazenadas em uma caixa postal, sendo processadas todas as vezes que se verifica a existência de correspondências.

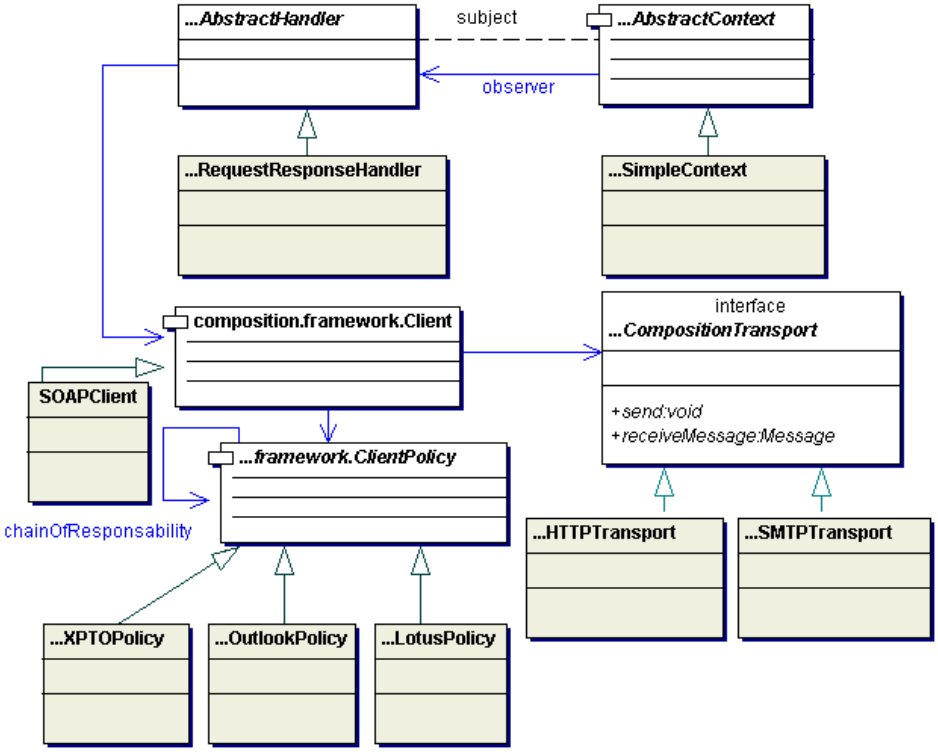

Figura 44 – Instância do *framework*: comunicação, transporte e *Context-Handler*

O serviço responsável por agendar os compromissos foi implementado utilizando uma versão anterior à do framework de composição apresentado (Ferreira et al., 2003). Para o tratamento de mensagens recebidas ao longo do processo de composição, foram implementadas sub-classes de *AbstractHandler*. Um exemplo de tratador implementado foi um *RequestResponseHandler*. Este tratador encapsula a chamada a um serviço de agenda. Além disto, o *RequestResponseHandler* trata as exceções que eventualmente podem acontecer no processo de chamada de um serviço de agenda.

Para o encapsulamento de informações relativas ao processo de marcação de um compromisso, foi estendido o contexto *SimpleContext*, que basicamente armazena as mensagens recebidas e a serem enviadas. Além disto, foi implementada a extensão *ClientSOAP* da classe *Client*, responsável pelo protocolo SOAP, viabilizando a conversação e o acesso a serviços. Para o entendimento

semântico foram implementadas conversões de informações para os serviços Lotus Notes®, Outlook® e XPTO, por meio de extensões da classe *ClientPolicy*.

4.4. Apoio ao Armazenamento e Catalogação de Artefatos de Software

Ambientes de desenvolvimento de software devem ser entendidos como um conjunto de ferramentas capazes de auxiliar a elaboração, a análise, o projeto, o desenvolvimento, o teste e a catalogação de componentes de software. Existem inúmeras alternativas capazes de atender a estes requisitos. Apesar de não mencionados, ambientes de desenvolvimento de software são fundamentais para a realização da camada de Artefatos de Software de ACCA.

Um exemplo de ambiente extensível para o desenvolvimento de aplicações é o Eclipse (Eclipse, 2003). Por meio deste ambiente, é possível adicionar, excluir ou alterar extensões de ferramentas, também chamadas de *plug-ins*. A partir destas extensões, é possível customizar o ambiente de desenvolvimento de acordo com as necessidades da equipe de desenvolvimento.

Na camada de Artefatos de Software de ACCA, a abstração mais relevante identificada foi o repositório de componentes de software. Um exemplo de realização de repositório é o CVS (CVS, 2002). O CVS (*Concurrent Versions System*) é uma estrutura utilizada para o controle de versões de artefatos de software. Além do controle de versão, é possível associar informações às unidades de software cadastradas neste tipo de repositório. O CVS apresenta uma linguagem própria para a consulta, alteração e atualização de artefatos de software em seu catálogo de informações. Um catálogo está associado ao repositório, onde fisicamente ficam armazenados os componentes de software.

É importante observar que este tipo de catálogo é recomendado para o desenvolvimento de software em equipe, armazenando uma gama de artefatos, inclusive arquivos fontes, não sendo, portanto uma proposta específica para o desenvolvimento de software baseado em componentes binários. Porém, isto não impede a sua utilização, já que é possível associar uma descrição detalhada a um artefato de software cadastrado em uma versão no repositório mantido pelo CVS. É importante observar que o repositório deve estar integrado ao ambiente de desenvolvimento, o que ocorre entre o Eclipse e o CVS.

Outros exemplos de catálogos que podem ser utilizados neste exemplo são repositórios que implementem UDDI (UDDI, 2002). UDDI (*Universal Description, Discovery and Integration*) é um padrão desenvolvido por um consórcio de empresas para a descoberta de serviços, equivalentes a componentes binários em execução. Nesta iniciativa, foi desenvolvido o conceito de *broker* de serviços. Um *broker* de serviços é capaz de receber e organizar descrições de diversos *Web Services* em categorias. Após armazenar e catalogar estes serviços, o *broker* é capaz de receber consultas por um serviço e retornar como resultado um conjunto de opções que atendam a essa descrição. Este mecanismo de envio e recuperação de descrição é definido por meio de um conjunto de especificações de mensagens XML (*Extensible Markup Language*) que definem como os *brokers* de serviços devem ser acessados. Desta forma, qualquer aplicação pode acessar um *broker* independente da plataforma em que esteja desenvolvida, desde que possua uma conexão de Internet com o *broker* de serviços.

Um repositório, que apresente estas características, é necessário, pois alguns artefatos de software utilizados já estão em execução distribuída, provendo um serviço remotamente. Este catálogo deve ser estendido para descrever a semântica e a sintaxe dos componentes. Além disto, deve-se considerar que estes artefatos se encontram distribuídos em ambientes de execução como provedores de serviços ou ainda servidores de aplicação. Exemplos destes ambientes de execução são servidores de aplicação como WAS (IBM-WAS, 2003), servidores .Net (MS-DotNet, 2003), dentre outros.

4.4.1. O Problema Exemplo: Realização da Camada Artefatos de Software

No problema exemplo apresentado, seis diferentes artefatos de software (Figura 45) precisam ser compostos e coordenados para a marcação de compromissos. Estes seis serviços estão desenvolvidos sob três tecnologias de componentes distintas: serviços Outlook®, serviços Lotus® e serviços XPTO.

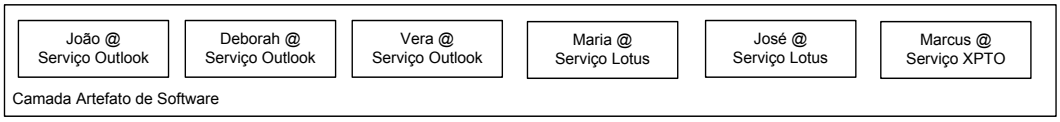

Figura 45 – Componentes de software disponíveis para a solução do problema

Além disto, estes serviços estão remotamente distribuídos em catálogos ou repositórios organizados por empresas (Figura 46), isto é, cada empresa dispõe de

um provedor de serviços ou servidor de aplicação aonde os artefatos estão acessíveis. Para auxiliar esta atividade, um grande catálogo ou repositório de referências pode ser criado, a partir do qual é possível catalogar e procurar pelos serviços de agenda dos usuários envolvidos armazenados em repositórios CVS ou UDDI.

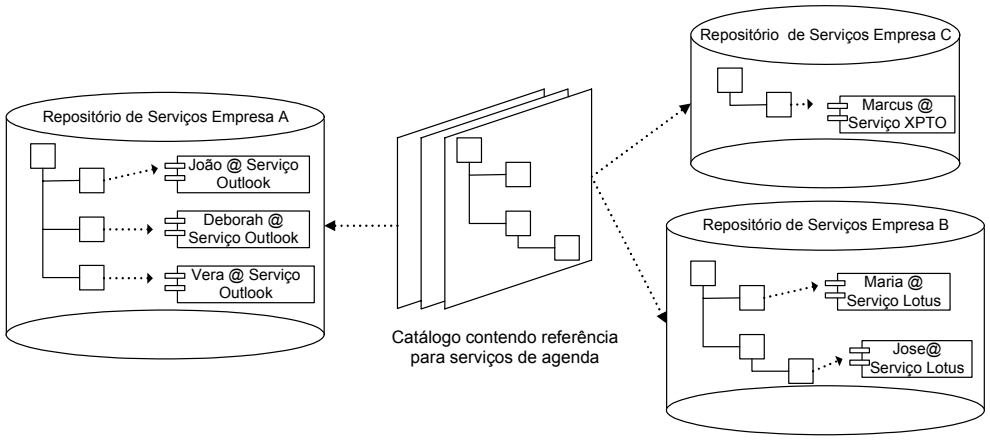

Figura 46 – Ilustração da realização conceitual da camada de artefatos de software