4
Avaliação

A avaliação do trabalho consiste em identificar as vantagens e desvantagens entre modelos de programação com abordagens orientadas a objetos tradicionais e abordagens baseadas na programação orientada a atributos

Para dar base ao estudo, foram desenvolvidas duas aplicaçõe utilizando o modelo de programação ASCS. A primeira, mais simples, implementa um sistema de notificação remota de logs, e o segundo, implementa um sistema de bate-papo que permite a conexão e comunicação de múltiplos usuários simultaneamente. A primeira seção deste capítulo apresenta estas aplicações.

Na segunda seção do capítulo, primeiro, é realizada uma análise qualitativa, comparando os dois trabalhos, utilizando o CDN (Cognitive Dimensions of Notations Framework) [21, 29]. Também foi realizada, através de uma abordagem quantitativa, uma análise referente à diferença de quantidade de linhas de código em tarefas semelhantes nos dois modelos de programação.

No final do capítulo avaliamos o impacto no desempenho causado pelo uso de um modelo de programação reflexivo baseado em anotações. Para isso, foi feita uma comparação quantitativa, entre os dois trabalhos, referente à diferença de desempenho para carga, conexão e acesso a componentes remotos.

4.1
Exemplos

Nesta seção são apresentadas as duas aplicações que foram desenvolvidas, para dar base ao estudo, utilizando o modelo de programação ASCS. A primeira, mais simples, implementa um sistema de notificação remota de logs, e o segundo, implementa um sistema de bate-papo que permite a conexão e comunicação de múltiplos usuários simultaneamente.

4.1.1
RemoteLogger

A aplicação *RemoteLogger*, desenvolvida originalmente com o modelo de programação original do SCS, oferece um mecanismo de realização de logs de

forma remota. Neste trabalho, a aplicação foi adaptada de forma a utilizar o novo modelo de programação ASCS.

Através da faceta disponibilizada pelo componente *LoggingComponent*, o usuário da aplicação informa a ocorrência de um log, atribuindo a mensagem e o tipo referente ao log. O tipo do log pode ser *warn*, *error*, ou *other*. Na ocorrência de um log, todos os observadores, que são implementados pelo componente *NotifyComponent* são notificados. Cada observador, então, adiciona os detalhes do log em um arquivo específico para o tipo de log ocorrido. A figura 4.1 ilustra o relacionamento entre os componentes.

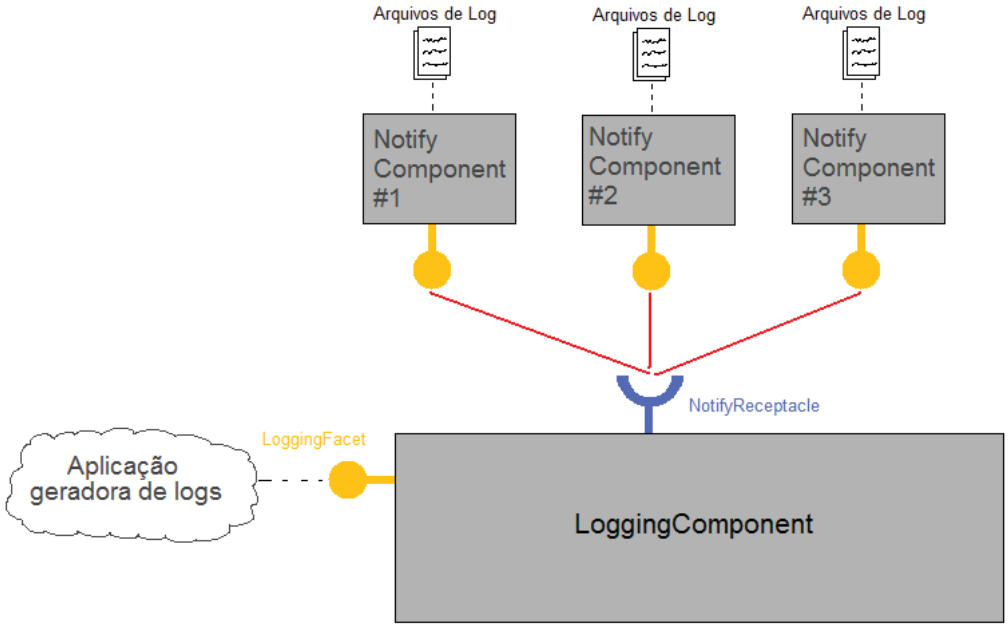

Figura 4.1: Componentes RemoteLogger

A seguir, é apresentado o arquivo IDL utilizado como base na aplicação.

Código 4.1: Arquivo IDL da aplicação RemoteLogger

```
1  module scs {
2   module demos {
3    module ascs {
4     module logger {
5      module generated {
6        enum LogType
7        {
8         WARN,
9         ERROR,
10        OTHER
11       };
12
13       valuetype LogMessage
14       {
15         public string message;
16         public LogType type;
17       };
18
```

```
19        interface Logging {
20          void printLogMsg(in string msg);
21          void printLog(in LogMessage msg);
22        };
23
24        interface Notify {
25          void warn(in string msg,in string date);
26          void error(in string msg,in string date);
27          void other(in string msg,in string date);
28        };
29      };
30    };
31   };
32  };
33 };
```

Onde:

* LogType

  Enumera os possíveis tipos de log. O sistema irá separar os logs em arquivos diferentes dependendo do tipo do log.

* LogMessage

  Estrutura que armazena o tipo do log e a mensagem correspondente.

* Logging

  Interface com a qual o usuário irá interagir, informando a ocorrência de um log. Internamente, o componente irá se comunicar com todas as facetas definida pela interface Notify, que funcionarão como observadores. Para cada observador será chamado o método relativo ao tipo de log ocorrido.

* Notify

  Interface que representa um observador da ocorrência de logs. A faceta relativa à esta interface deve adicionar os detalhes do log em um arquivo específico dependendo do tipo de log ocorrido.

O trecho de código A.1 definido no apêndice desta dissertação apresenta o código fonte completo do componente *LoggingComponent*, que disponibiliza uma faceta seguindo a interface *Logging* definida na IDL.

O trecho de código A.2 apresenta o código fonte completo do componente *NotifyComponent*, que disponibiliza uma faceta seguindo a interface *Notify* definida na IDL.

Os arquivos de saída gerados por cada *NotifyComponent* apresentam o formato mostrado na figura 4.2.

Código 4.2: Arquivo gerado pelo componente NotifyComponent

```
1 [Fri May 14 15:36:09 GMT-03:00 2010] [Component aNotify (v1.0.0) - Java
    platform :: Component aLogging (v1.0.0) - Java platform received: ERROR
     Test.
2 [Fri May 14 15:36:09 GMT-03:00 2010] [Component aNotify (v1.0.0) - Java
    platform :: Component aLogging (v1.0.0) - Java platform received: ERROR
     Test.
3 [Fri May 14 16:57:33 GMT-03:00 2010] [Component aNotify (v1.0.0) - Java
    platform :: Component aLogging (v1.0.0) - Java platform received: ERROR
     Test.
```

4.1.2 Chat

A aplicação *Chat*, desenvolvida originalmente neste trabalho, implementa um sistema de bate-papo que permite a conexão e comunicação de múltiplos usuários simultaneamente.

O sistema é composto por dois componentes. O componente *MediatorComponent* implementa um mediador, que é responsável por receber mensagens de um determinado usuário do bate-papo e repassar para os demais usuários conectados. O componente *UserComponent* representa um usuário do bate-papo, que se conecta com o mediador, enviando e recebendo mensagens.

A figura 4.2 mostra como os componentes do sistema se relacionam.

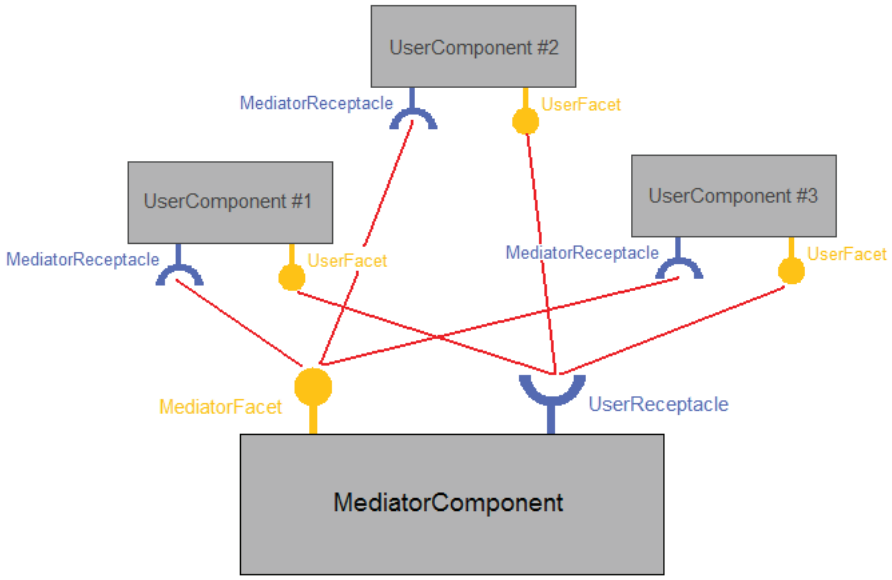

Figura 4.2: Componentes Chat

A figura 4.3 apresenta o arquivo IDL que foi utilizado como base para a implementação da aplicação.

Código 4.3: Arquivo IDL - Chat

```
1 module scs {
2   module demos {
```

```
3   module ascs {
4    module chat {
5     module generated{
6
7       interface User {
8         void receiveMessage(in string sender, in string msg);
9         void receiveWhisper(in string sender, in string msg);
10        void receiveJoinNotification(in string user);
11        void receiveLeaveNotification(in string user);
12        string getName();
13      };
14
15      interface Mediator {
16        typedef sequence <string> string_seq;
17        void sendMessage(in string sender, in string msg);
18        void sendWhisper(in string sender, in string receiver, in string
              msg);
19        void sendJoinNotification(in string user);
20        void sendLeaveNotification(in string user);
21        string_seq getUserNames();
22      };
23
24    };
25   };
26  };
27 };
28 };
```

Onde:

* User.receiveMessage

  O usuário recebe uma nova mensagem que foi enviada para todos os usuários presentes na sala, atualizando sua interface.

* User.receiveWhisper

  O usuário recebe uma nova mensagem privada.

* User.receiveJoinNotification

  O usuário recebe uma notificação de que um novo usuário entrou na sala de bate-papo.

* User.receiveLeaveNotification

  O usuário recebe uma notificação de que um usuário saiu da sala de bate-papo.

* User.getName

  Retorna o nome do usuário.

* Mediator.sendMessage

  Repassa para todos os usuários presentes na sala a ocorrência de uma nova mensagem pública

* Mediator.sendWhisper

  Envia uma mensagem para um usuário específico

* Mediator.sendJoinNotification

  Notifica todos os usuários presentes na sala que um novo usuário se conectou.

* Mediator.sendLeaveNotification

  Notifica todos os usuários presentes na sala que um determinado usuário saiu da sala.

* Mediator.getUserNames

  Obtém o nome de todos os usuários presentes

A figura A.3 definida no apêndice desta dissertação apresenta o código-fonte completo do componente *MediatorComponent*. A figura A.4 apresenta o código-fonte completo do componente *UserComponent*. A figura A.5 mostra o código-fonte completo da classe *SendAction*, definida no contexto do usuário da sala de bate-papo. Esta classe é responsável por enviar uma nova mensagem para todos os usuários do *chat*, ou para um especificamente, acessando o mediador do *chat*.

A figura 4.3 mostra três instâncias da aplicação sendo executadas. No título de cada janela é exibido o nome do usuário relativo à instância. O componente de texto central exibe os eventos ocorridos e as mensagens de texto recebidas e enviadas. O componente da direita lista os usuários presentes na sala. A opção "whisper"permite que uma mensagem reservada seja enviada ao usuário selecionado.

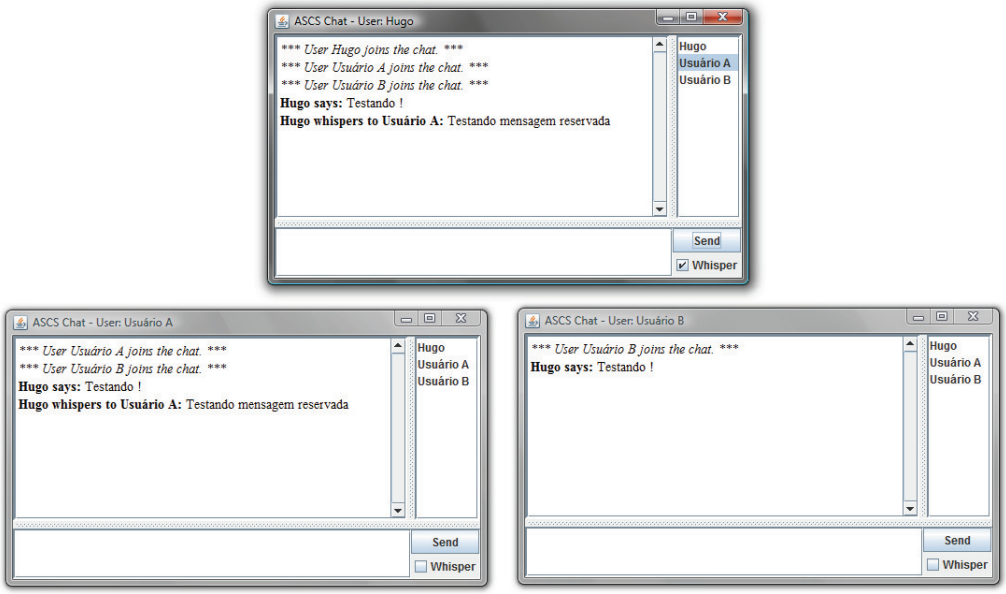

Figura 4.3: Chat

4.2 Análise dos Modelos de Programação

Nesta seção, primeiro, é realizada uma análise qualitativa, comparando os dois trabalhos, utilizando o CDN (Cognitive Dimensions of Notations Framework) [21]. Também foi realizada, através de uma abordagem quantitativa, uma análise referente à diferença de quantidade de linhas de código em tarefas semelhantes nos dois modelos de programação.

4.2.1 Avaliação Qualitativa (CDN)

Em um modelo de pesquisa qualitativa, diferentemente de uma abordagem quantitativa, não se pretende chegar a conclusões absolutas. Nesta forma de estudo procura-se atacar pontos nem sempre mensuráveis, utilizando diferentes critérios de avaliação para cada um deles. Um estudo baseado em técnicas qualitativas foca na identificação de aspectos relevantes da usabilidade do software.

O CDN (Cognitive Dimensions of Notations Framework) foi inicialmente idealizado com o objetivo de avaliar linguagens visuais através da definição de 14 dimensões. Neste trabalho, atribuímos interpretações a estas dimensões para adequá-las à avaliação de API's, e então utilizar o CDN em nosso estudo através de uma abordagem qualitativa. As dimensões avaliação progressiva, decisões prematuras, notações secundárias, e operações complexas, foram consideradas irrelevantes para o nosso estudo e descartadas.

Para aplicar o CDN devemos identificar as tarefas representativas do sistema a serem estudadas, executá-las tendo em vista cada uma das dimensões, e por fim, compará-las com base nas dimensões.

Em [30], podemos encontrar os conceitos genéricos de cada uma das dimensões do CDN. Para cada uma destas dimensões, atribuiremos uma intepretação específica para que ela ganhe significado no processo de avaliação de um modelo de programação, e então, compararemos o modelo de programação original do SCS com o ASCS, tomando como base a tarefa que consiste no desenvolvimento de um componente.

Viscosidade (Viscosity)

A viscosidade é a resistência a mudanças. Ou seja, a viscosidade é referente ao custo relativo à mudanças necessárias no código para adaptá-lo a um uso diferente daquele para o qual foi projetado. Ao incluir, por exemplo, uma nova funcionalidade em um modelo de programação com uma viscosidade alta, várias entidades do modelo são impactadas, alterando significativamente seu uso.

No modelo de programação original do SCS, mudanças no componente podem acarretar mudanças na fábrica, e também na forma de acessar, por exemplo, um receptáculo.

O trecho de código 2.4, apresentado anteriormente no capítulo 3, nos mostra a implementação de uma fábrica SCS. Se, por exemplo, incluirmos novas facetas ou novos receptáculos no componente, a fábrica precisará ser alterada, junto com a própria implementação do componente.

No ASCS, qualquer mudança estrutural do componente será feita alterando apenas o código de implementação do componente. Sendo assim, o modelo ASCS oferece uma viscosidade mais baixa se comparado com o modelo de programação original do SCS. É importante ressaltar que essa diferença é relativa à forma como o SCS original foi implementado, não sendo necessariamente devidas ao uso de anotações.

O fato do modelo de programação original do SCS forçar o usuário a misturar código funcional com código relacionado à tecnologia e infra-estrutura utilizada também pode ser considerado um problema de viscosidade, já que a implementação do componente fica atrelada ao framework SCS.

Visibilidade (Visibility)

A visibilidade se traduz na facilidade de visualizar todas as características relacionadas a uma mesma entidade. Um exemplo de problema de visibilidade em um modelo de programação é a necessidade de acesso a um valor ou conceito que não é claramente visível do ponto do programa onde se precisa utilizá-lo.

No modelo de programação original do SCS as diferentes partes que compõem um mesmo componente estão espalhadas em trechos de código e arquivos separados. Por exemplo, é necessária a codificação de uma fábrica separada responsável por relacionar ao componente todas as descrições de suas facetas e receptáculos. As relações entre os elementos que fazem parte de um mesmo componente não é facilmente identificada no modelo de programação original do SCS.

O trecho de código 2.3, apresentado anteriormente no capítulo 3, nos mostra que para acessar um receptáculo do componente é necessária a utilização de um identificador textual que está definido no arquivo da fábrica, o que introduz um problema de visibilidade. Neste caso, o programador precisaria conhecer a implementação deste outro arquivo para poder acessar o receptáculo. Além deste problema, no modelo de programação original do SCS, as facetas do componente podem ser facilmente visualizadas no próprio código de implementação do componente, porém os receptáculos são definidos apenas na fábrica, o que introduz uma dificuldade na visualização do conjunto completo de facetas e receptáculos do componente.

No ASCS, o uso de anotações próximas aos elementos do código que representam um conceito do modelo de componentes simplificam o aspecto de visibilidade. Na tarefa de criação de um componente, o único arquivo de código que o programador precisa desenvolver é o arquivo de implementação do próprio componente.

Porém, o modelo de programação ASCS também apresenta alguns problemas de visibilidade. O motivo de certas restrições impostas, como por exemplo, a necessidade de acrescentar métodos *getter* e *setter* para receptáculos não é intuitivo do ponto de vista do desenvolvedor.

Dispersão (Diffusiness)

A dispersão é referente à verbosidade. Um modelo de programação com alto nível de dispersão requer um trabalho extenso, com a necessidade de escrita de uma grande quantidade de texto, para tarefas que poderiam ser mais simples.

Uma característica inerente à programação orientada a atributos é a baixa verbosidade. Consequentemente, o ASCS reduz a quantidade de linhas de código necessárias para codificar um componente. Os únicos elementos utilizados no ASCS para relacionar o código ao framework, são as anotações (uma para cada conceito do modelo de componentes). No modelo original do SCS o desenvolvedor precisa interagir diretamente com as classes do framework.

Dependências ocultas (Hidden dependencies)

Uma dependência oculta em um modelo de programação é quando, para se realizar uma determinada tarefa, há uma dependência de uma entidade ou de um conceito que não está totalmente explícita. Ou seja, o desenvolvedor precisa conhecer os detalhes de implementação do *framework* para deduzir as implicações do uso de sua API.

Nesse aspecto, as mesmas características apresentadas na dimensão *Visibilidade* também podem ser consideradas problemas de dependênca oculta. Por exemplo, ao criar o *servant* de um componente, como exemplificado no trecho de código 2.3 apresentado no capítulo 3, o desenvolvedor deve saber que ele precisa criar um construtor que recebe o objeto de contexto, além de ter que estender a classe *POA* relativa. O ASCS também possui restrições do mesmo tipo, porém, elas são verificadas em tempo de compilação, deixando assim de serem consideradas ocultas.

O ASCS não é conceitualmente independente de CORBA [25, 26]. Vimos que em momentos específicos o desenvolvedor do componente precisa fazer referência a interfaces geradas pelo compilador de IDL. Estas interfaces são interfaces java "puras", ou seja, não possuem nenhum tipo de referência a CORBA ou a qualquer outra tecnologia. Porém, como são interfaces específicas geradas pelo compilador de IDL, pode-se considerar que existe uma dependência conceitual oculta.

Abstrações (Abstractions)

Esta dimensão é relativa à quantidade de abstrações com as quais o desenvolvedor tem que interagir para executar uma determinada tarefa.

Um dos objetivos do uso de anotações no modelo de programação ASCS é diminuir o número de abstrações que o desenvolvedor precisa interagir. No ASCS, as abstrações que o desenvolvedor do componente precisa entender e usar são as abstrações do próprio modelo de componentes (facetas, re-

ceptáculos, etc) que são traduzidas de forma direta para as anotações. Como dito no item *Dispersão*, no modelo original do SCS o desenvolvedor tem que interagir com várias abstrações, que se traduzem a classes do framework, sendo que muitas não tem relação direta com elementos do modelo de componentes.

Predisposição a erros (Error-Proneness)

Um modelo de programação é predisposto a erros quando uma determinada tarefa pode ser executada de forma errada pelo desenvolvedor sem que ele perceba de forma imediata, ou ainda quando características do próprio modelo de programação induzem o desenvolvedor ao erro. A predisposição a erros está intimamente ligada à existência de dependências ocultas e à distância semântica das entidades do modelo de programação com o domínio do problema.

No ASCS, os possíveis erros relacionados a estrutura do componente e uso das anotações são detectados em tempo de compilação. No modelo de programação original do SCS existem várias possibilidades de erros que serão detectadas apenas em tempo de execução, como por exemplo a informação de um nome errado na hora de obter um receptáculo, a não-inclusão de um construtor que recebe o contexto, ou o esquecimento de estender a classe *POA* correspondente a cada faceta.

Proximidade com o domínio (Role-Expressiveness / Closeness of Mapping / Consistency)

A proximidade com o domínio é referente à distância semântica das entidades do modelo de programação com o domínio do problema. Quando temos, por exemplo, uma classe cujo nome não representa de fato seu papel, temos uma baixa proximidade com o domínio.

No SCS, cujo modelo de programação é basicamente OO, encontramos um distanciamento maior devido às limitações encontradas na própria linguagem de programação, que não possui elementos que representam, por exemplo, facetas e receptáculos. As anotações definidas no ASCS fazem um papel de "cola"entre os elementos do código-fonte do componente com os conceitos do modelo de componentes.

Além disso, no SCS vemos a inexistência de classes que representem facetas ou receptáculos. No ASCS, cada anotação marca diretamente um conceito do modelo de componentes, onde por exemplo a anotação *@Receptacle* marca um elemento que representa de fato o receptáculo, e a anotação

@Property marca um elemento que representa de fato uma propriedade do componente.

Provisionamento (Provisionality)

O provisionamento é o grau de comprometimento com as ações ou marcações. Um modelo de programação que apresentar uma maior completude ou flexibilidade será considerado de maior provisionamento.

Nesta dimensão, o modelo de programação original do SCS leva vantagem, pois permite uma maior flexibilidade. O modelo de programação original do SCS, apresenta, como parte de sua API, classes que disponibilizam acesso direto ao middleware(*CORBA [25, 26]*) utilizado, o que permite a realização de operações não esperadas de mais baixo nível, permitindo um controle maior do middleware. Nestes casos extremos, o programador também pode fazer acesso direto a estas classes. Porém, esta prática iria contra às principais motivações do ASCS.

O fato do ASCS possuir uma menor flexibilidade não é necessariamente um ponto negativo. A API de anotações proporcionada pelo ASCS é uma API de mais alto nível. API's de mais alto nível proporcionam, naturalmente e propositalmente, menor flexibilidade, podendo assim ser mais simples e fácil de usar.

4.2.2 Avaliando Outras Abordagens

Na comparação utilizando as dimensões do CDN, podemos ver que a necessidade de definir a composição e a estrutura do componente no próprio código orientado a objetos introduz diversos problemas de uso do modelo de programação. Como vimos, a definição de um modelo de programação orientado a atributos aborda este problema isolando estas informações em uma camada de anotações. Para este caso, vários outros tipos de abordagens também seriam possíveis.

A utilização da orientação a aspectos [8] é uma outra possível alternativa para este problema de forma a isolar tarefas, como a definição de quais facetas e receptáculos compõem o componente, em uma camada de aspectos. Comparando com a abordagem orientada a atributos desenvolvida neste trabalho, esta idéia seria mais vantajosa quando comparamos pela dimensão *Viscosidade*, já que o código de implementação do componente ficaria totalmente independente da camada de aspectos, e na abordagem orientada a atributos, anotações e código funcional ainda permanecem no mesmo arquivo de código. Porém, a

tecnologia de orientação a aspectos não possui suporte nativo pela linguagem. Isto ocasionaria a necessidade da utilização de uma biblioteca terceira que ofereça suporte a aspectos. Além disso, a utilização de aspectos introduz problemas de *Visibilidade*, já que não existe nenhuma ligação a partir da camada orientada a objetos com a camada de aspectos.

Uma abordagem declarativa, de forma a utilizar um arquivo de configurações externo ao código para definir a composição e a estrutura do componente, também é uma alternativa. Neste caso, a principal vantagem é relativa à dimensão *Consistência*, já que não misturaríamos a orientação a objetos com nenhum outro paradigma. Porém, esta abordagem apresenta o mesmo problema relativo à *Visibilidade* encontrado em uma possível abordagem orientada a aspectos, já que também não existe nenhuma ligação a partir da camada orientada a objetos com os arquivos de configuração do componente.

4.2.3 Quantidade de Linhas de Código

Devido ao fato do modelo de programação do ASCS ser composto por anotações, o desenvolvedor do componente não precisa instanciar classes e chamar métodos para realizar a integração com a infra-estrutura de componentes. Sendo assim, a quantidade de linhas de código necessárias para criar um componente utilizando o ASCS tende a ser menor.

Os exemplos *HelloPrinter* e *RemoteLogger* foram utilizados para realizar esta comparação.

A tabela 4.1 mostra a quantidade de linhas de código que foram necessárias pra desenvolver os componentes de cada um dos exemplos.

	HelloPrinter	**RemoteLogger**
SCS	103 linhas	174 linhas
ASCS	21 linhas	96 linhas

Tabela 4.1: Quantidade de Linhas de Código

A diferença de linhas de código necessárias tende a aumentar ainda mais quando temos uma grande quantidade de receptáculos no componente ou quando temos uma grande quantidade de componentes.

4.3 Análise de Desempenho

A abordagem orientada a atributos adotada impõe uma camada extra de anotações que envolve um processamento extra. A importância da avaliação desta seção é determinar se o *overhead* gerado por esse processamento extra

é ou não significativo. Para isso, foi feita uma análise comparativa de desempenho, onde a carga, conexão, e acesso remoto a componentes foram testados nas diferentes circunstâncias.

A tabela 4.2 mostra a média, obtida em 30 testes, do tempo gasto em milisegundos para a carga, conexão, e acesso à faceta do componente *HelloPrinter*. A variância não passou de 10%. Os testes abaixo foram realizados em ambiente local, em uma CPU *Intel Core2 Quad Q6600*.

	carga	**conexão**	**chamada**
SCS	10 ms	12 ms	5 ms
ASCS	30 ms	15 ms	5 ms

Tabela 4.2: Desempenho HelloPrinter SCS

Pode-se perceber que a única operação que oferece um *overhead* proporcionalmente significativo é a operação de carga. Este *overhead* tende a aumentar ainda mais para componentes com elevado número de facetas e receptáculos. Porém, deve-se considerar que a operação de carga é executada apenas uma vez, na hora da implantação do componente. Deve-se considerar também que a diferença absoluta de tempo (na casa de milisegundos) normalmente não oferecerá uma preocupação real em termos de desempenho no momento da implantação do componente.

A operação de chamada ou acesso à faceta do componente não sofreu alterações de desempenho no novo ASCS.