Fundação CECIERJ - Vice Presidência de Educação Superior a Distância

Curso de Tecnologia em Sistemas de Computação
Disciplina de Arquitetura e Projeto de Sistemas II
Gabarito - AD2 1° semestre de 2009.

Nome –

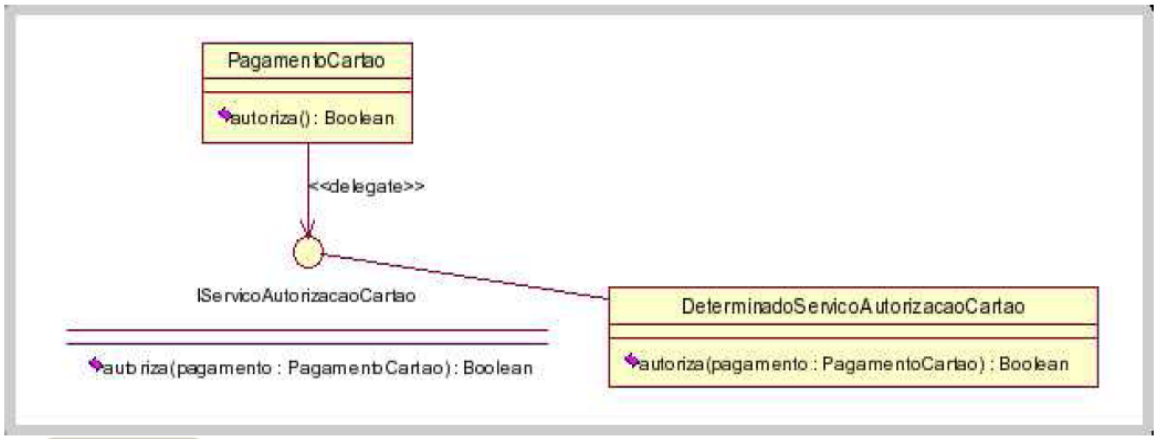
Observações:

1. Prova com consulta.

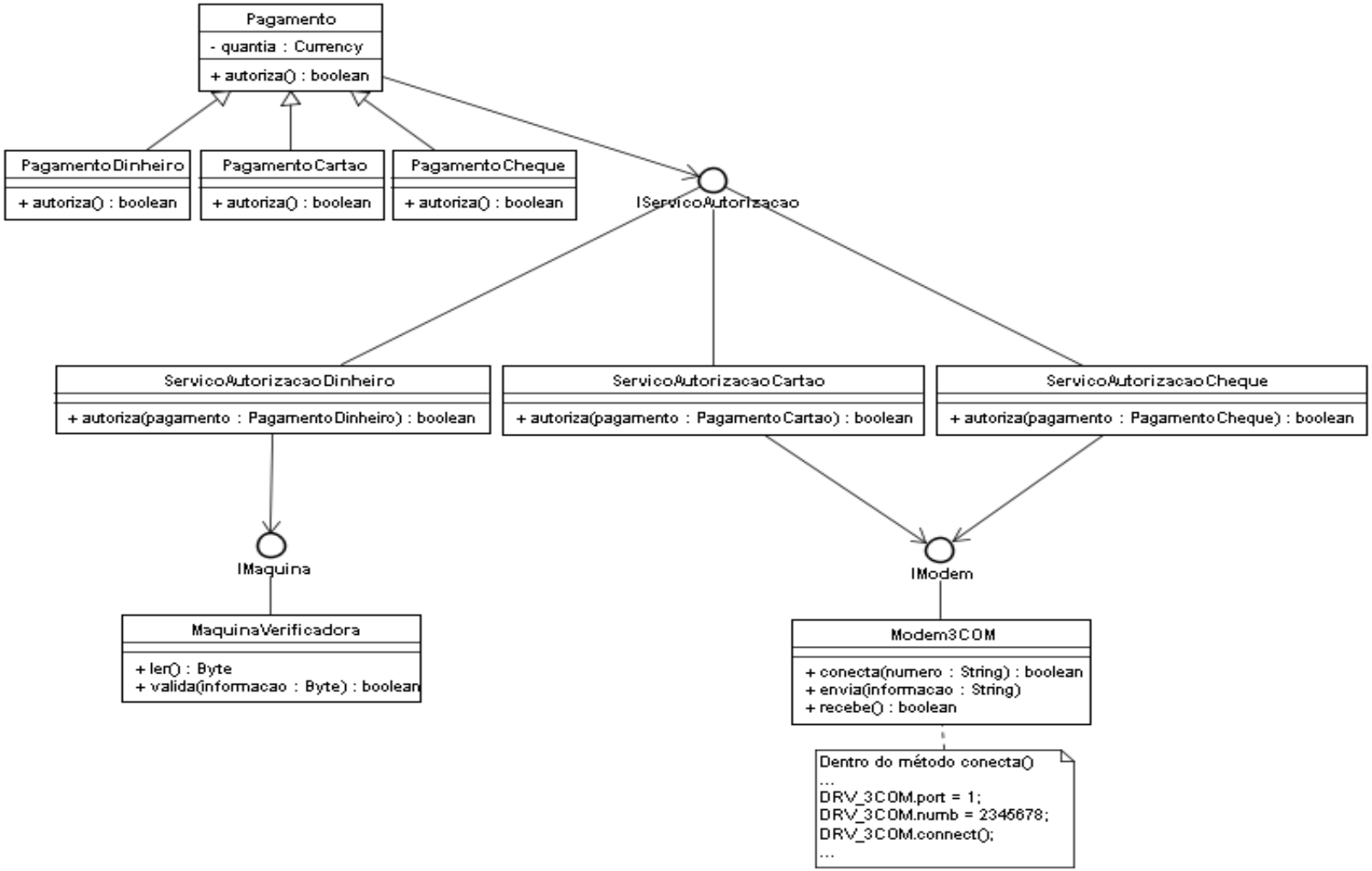
Atenção: Como a avaliação à distância é individual, caso sejam constatadas semelhanças entre provas de alunos distintos, será atribuída a nota ZERO a TODAS as provas envolvidas. As soluções para as questões podem ser buscadas por grupos de alunos, mas a redação final de cada prova tem que ser distinta. ALÉM DISSO, às questões desta AD respondidas de maneira muito semelhantes às respostas oriundas dos gabaritos já publicados de ADs de períodos anteriores, será atribuída a nota ZERO, incluindo também cópias diretas e sem sentido de tópicos dos slides das aulas.

Questão 1 [1 ponto]

Evolua o modelo abaixo para contemplar o padrão *Indirection* para as três formas de pagamento *cartão*, *cheque* e *dinheiro*. Nesse caso, a indireção deve ser feita para o modem (cartão e cheque) e para a máquina de verificação de autenticidade de notas (dinheiro). Sobre o modelo gerado, perceba que um determinado serviço de autorização pode ser visto como um componente (interfaces providas e requeridas).



Resposta:


Pagamento
- quantia : Currency
+ autoriza() : boolean
PagamentoDinheiro
+ autoriza() : boolean
PagamentoCartao
+ autoriza() : boolean
PagamentoCheque
+ autoriza() : boolean
IServicoAutorizacao
ServicoAutorizacaoDinheiro
+ autoriza(pagamento : PagamentoDinheiro) : boolean
ServicoAutorizacaoCartao
+ autoriza(pagamento : PagamentoCartao) : boolean
ServicoAutorizacaoCheque
+ autoriza(pagamento : PagamentoCheque) : boolean
IMaquina
IModem
MaquinaVerificadora
+ ler() : Byte
+ valida(informacao : Byte) : boolean
Modem3COM
+ conecta(numero : String) : boolean
+ envia(informacao : String)
+ recebe() : boolean
Dentro do método conecta()
...
DRV_3COM.port = 1;
DRV_3COM.numb = 2345678;
DRV_3COM.connect();
...

Questão 2 [3 pontos]

O padrão *Facade* visa prover uma interface única para um conjunto de interfaces de um subsistema, facilitando o seu uso.

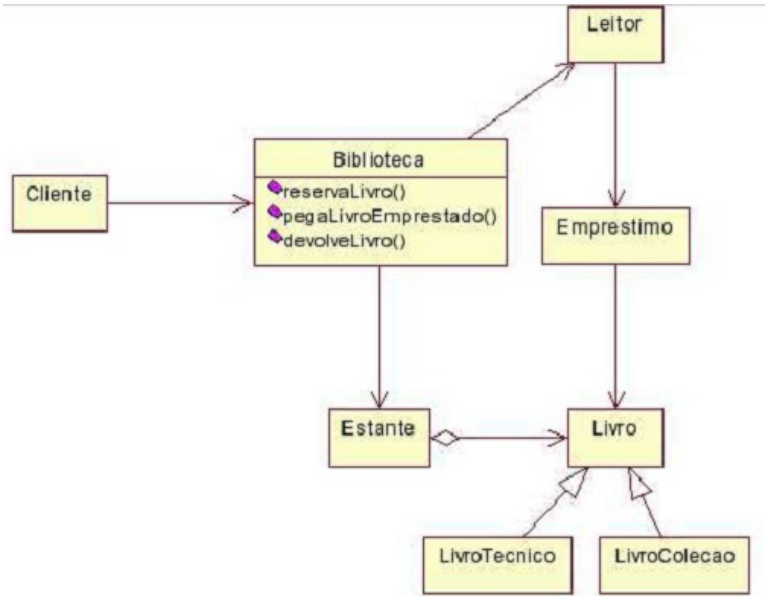
a) [1 ponto] Descreva um exemplo de situação onde esse padrão é útil.
b) [1 ponto] Quais são os benefícios alcançados com o uso desse padrão?
c) [1 ponto] Desenhe um modelo de classes exibindo como o padrão pode ser utilizado na situação descrita.

Resposta:

a) Um possível exemplo de situação onde esse padrão é útil pode ser verificado no projeto da interação entre um sistema cliente com um sistema de controle de bibliteca. Neste caso, visando prover uma interface única para um conjunto de interfaces de um subsistema, um conjunto de classes (e.g., `Estante`, `Leitor`, `Empréstimo` e `Livro`) deve se comportar como um componente, de maneira que o cliente deva falar com uma única interface (i.e., `Biblioteca`). Dessa forma, operações como `reservaLivro()`, `pegarLivroEmprestado()` e `devolveLivro()` estariam concentradas na classe `Biblioteca`, facilitando seu uso, compreensão e manutenção.

b) Dentre os benefícios alcançados com a utilização do padrão *Facade*, pode-se citar: (i) separa a responsabilidade da criação complexa em objetos auxiliares coesos; (ii) oculta a lógica de criação potencialmente complexa; e (iii) permite a introdução de estratégias de gerenciamento de memória para a melhoria do desempenho, como o uso de *cache* ou reciclagem de objetos.

c) Modelo de classes correspondente à situação apresentada em (a):



Questão 3 [2 pontos]

Suponha, por exemplo, que seja necessário um apoio para salvar instância de uma classe `Venda` (de um sistema de PDV) em um banco de dados. Pelo padrão *Information Expert*, atribuir essa responsabilidade à própria classe `Venda` tem alguma justificativa, pois a venda tem os dados que precisam ser salvos.

a) [1 ponto] Aponte e explique pelo menos três implicações decorrentes dessa decisão de projeto.
b) [1 ponto] Após refletir sobre as implicações respondidas em (a), aponte e explique uma solução mais razoável que leve em consideração outros padrões de projeto.

Resposta:

a) Implicações: (i) a tarefa exige um número relativamente grande de operações de apoio relacionadas ao banco de dados, nenhuma delas voltada ao conceito de *venda*, de modo que a classe `Venda` se torna não-coesa; (ii) a classe `Venda` precisa estar acoplada à interface do banco de dados relacional (como JDBC em tecnologias Java), de modo que seu acoplamento aumenta – além disso, o acoplamento não é sequer relacionado a outro objeto do domínio, mas a um tipo em particular de interface de banco de dados; e (iii) salvar objetos em um banco de dados relacional é uma tarefa geral, para a qual muitas classes precisam de apoio – colocar essas responsabilidades na classe `Venda` sugere que haverá reuso inadequado ou muita duplicação em outras classes que fazem a mesma coisa. Concluindo: mesmo que, de acordo com o padrão *Information Expert*, `Venda` seja uma candidata lógica para salvar a si própria em um banco de dados, ela leva a um projeto com coesão baixa, acoplamento alto e baixo potencial de reuso.

b) Uma solução razoável é criar uma nova classe que seja a única responsável por salvar objetos em algum tipo de meio de armazenamento persistente, como por exemplo, um banco de dados relacional, denominando-a de `ArmazenamentoPersistente`. Essa classe é uma criação da imaginação (*Pure Fabrication*). Observe o nome: `ArmazenamentoPersistente`. Esse é um conceito inteligível, apesar de o nome ou conceito de "armazenamento persistente" não ser algo que se encontra no Modelo de Domínio. Além disso, se um projetista perguntasse a um vendedor em uma loja "Você trabalha com objetos de armazenamento persistente?", ele não entenderia. Ele entende conceitos como "venda" e "pagamento". `ArmazenamentoPersistente` não é um conceito do domínio, mas algo criado ou inventado por conveniência do desenvolvedor de software. A aplicação do padrão *Pure Fabrication* resolve os seguintes problemas de projeto: (i) `Venda` permanece bem projetada, com coesão alta e acoplamento baixo (*Low Coupling*); (ii) a própria classe `ArmazenamentoPersistente` é relativamente coesa, tendo o único objetivo de armazenar ou inserir objetos em um meio de armazenamento persistente; e (iii) a classe `ArmazenamentoPersistente` é um objeto muito genérico e reutilizável. A criação de uma "invenção pura", nesse caso, é exatamente a situação em que seu uso é necessário – eliminar um projeto ruim baseado no padrão *Information Expert*,

com coesão e acoplamento inadequados, por um bom projeto no qual há maior potencial de reuso. Note que a ênfase está na alocação das responsabilidades (como acontece com todos os padrões GRASP). No caso apresentado, as responsabilidades são transferidas da classe Venda (motivada pelo padrão *Information Expert*) para uma "invenção pura" (motivada pelo padrão *Pure Fabrication*), visando alta coesão e baixo acoplamento (motivado pelo padrão *Low Coupling*).

Questão 4 [2 pontos]

Sob o risco de simplificar o conceito de *frameworks*, o pesquisador Larman descreve brevemente um *framework* como "um conjunto de objetos *extensível* para funções relacionadas" e aponta como exemplo mais significativo um *framework* de GUI (*Graphical User Interface*). A fim de expandir a simplificação feita por Larman, aponte pelo menos cinco características que sedimentem o conceito de *frameworks* e exemplifique este conceito por meio de um framework de GUI utilizado na indústria de software. Dica: explorar conceitos de *interfaces*, *classes e métodos concretos e abstratos* e *mensagem*.

Resposta:

Considerando a situação apresentada, a qualidade distinta de um *framework* é que ele fornece uma implementação para as funções básicas e invariantes, e inclui um mecanismo para permitir que o desenvolvedor se conecte às diversas funções ou as estenda. Em geral, um *framework*: (i) é um conjunto coeso de interfaces e classes que colaboram para fornecer serviços para a parte básica e constante de um subsistema lógico; (ii) contém classes concretas e (especialmente) abstratas, que definem interfaces a serem seguidas, interações entre objetos das quais participar e outros invariantes; (iii) em geral, (mas não necessariamente) exige que o usuário do *framework* defina subclasses das classes existentes no mesmo, para fazer uso, personalizar e estender os seus serviços; (iv) possui classes abstratas que podem conter métodos abstratos e concretos; e (v) depende do *Princípio de Hollywood*: "não ligue para nós, nós ligaremos para você" – isso significa que as classes definidas pelo usuário (e.g., novas subclasses) receberão mensagens das classes pré-definidas do *framework* e, normalmente, elas são tratadas usando a implementação de métodos abstratos da superclasse. Ademais, os *frameworks* oferecem um alto grau de reutilização. Por exemplo, o *framework GUI Swing* da linguagem Java fornece muitas classes e interfaces para funções básicas de GUI. Os desenvolvedores podem adicionar dispositivos especializados, estabelecendo subclasses das classes *Swing* e sobrepondo certos métodos. Os desenvolvedores também podem se conectar a vários comportamentos de resposta a eventos por meio de classes de dispositivos predefinidos (e.g., *JButton*), registrando receptores ou assinantes, com base no padrão *Observer*.

Questão 5 [1 ponto]

Pesquise acerca da arquitetura de interface com o usuário MVC (*Model-View-Controller*) e explique seus princípios e principais componentes, ilustrando o relacionamento entre

eles. Além disso, cite um *framework* utilizado na indústria de software que implemente a arquitetura MVC, explicando seus fundamentos e funcionalidades.

Resposta:

De modo geral, estilos arquiteturais servem para dar uma diretriz de como a arquitetura do sistema deve ser construída. A Engenharia de software, diferentemente das demais engenharias, não está sujeita a "leis da natureza". Então, estilos arquiteturais são "regras artificiais" que devem ser seguidas pelos desenvolvedores. Um dos principais estilos arquiteturais é o MVC (*Model-View-Controller*). A filosofia básica do MVC é que as classes do sistema devem ser separadas em três tipos: modelo, visão e controle

As classes do tipo "modelo" são classes derivadas do processo de análise, que representam os principais conceitos do domínio. Essas classes são usualmente persistidas em banco de dados. Por outro lado, as classes do tipo "visão" são classes criadas durante o projeto para fazer a interface com o usuário propriamente dita. Normalmente essas classes manipulam classes de modelo. Finalmente, as classes do tipo "controle" são classes que fazem a orquestração, ou seja, a recepção de eventos externos e o acesso às classes de modelo e visão.

A Figura 1 apresenta a relação entre esses tipos de classe. A partir dessa figura, é possível identificar algumas regras (que exerce o papel das leis da natureza para outras áreas do conhecimento): (1) classes de modelo não conhece ninguém (a não ser outras classes de modelo), (2) classes de visão conhece somente classes de modelo (e outras classes de visão), e (3) classes de controle conhece tanto classes de modelo quanto classes de visão (e outras classes de controle).

Com a utilização do MVC, alguns benefícios podem ser observados: separação de responsabilidades clara, alto grau de substituição da forma de interface com o usuário, alto grau de reutilização das entidades do domínio (classes de modelo), e possibilidade de múltiplas interfaces com o usuário, trabalhando de forma simultânea (modo texto, janelas, web, celular, etc.).

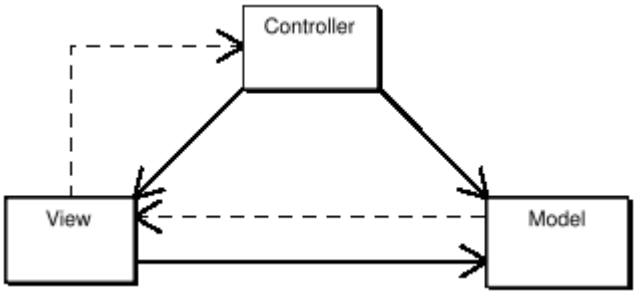


Figura 1 – Representação da arquitetura *MVC*

Um dos *frameworks* que implementa a arquitetura MVC, bastante utilizado pela indústria, é o *Struts*. O *Struts* é um *framework* que provê uma implementação do chamado *Modelo 2* ou *MVC-2* (variante do MVC oficializada pela *Sun*). O *MVC-2* é uma

arquitetura de desenvolvimento *web* que utiliza *servlets* e páginas *JSP* (*JavaServer Pages*) na mesma aplicação: os *servlets* são responsáveis pelo acesso aos dados e pelo fluxo navegacional, ao passo que as páginas *JSP* tratam da apresentação. Isso permite que desenvolvedores Java e desenvolvedores HTML trabalhem cada um em sua parte da aplicação. Nesse sentido, o *Struts* fornece um *servlet* controlador para lidar com o fluxo navegacional e as classes especiais para ajudar com o acesso dos dados. Uma biblioteca de *tags* personalizadas faz parte do *framework* a fim de aumentar a facilidade de uso do *Struts* considerando as páginas *JSP*. Na Figura 2, apresenta-se um exemplo no qual as etapas são descritas a seguir, de acordo com a numeração indicada:

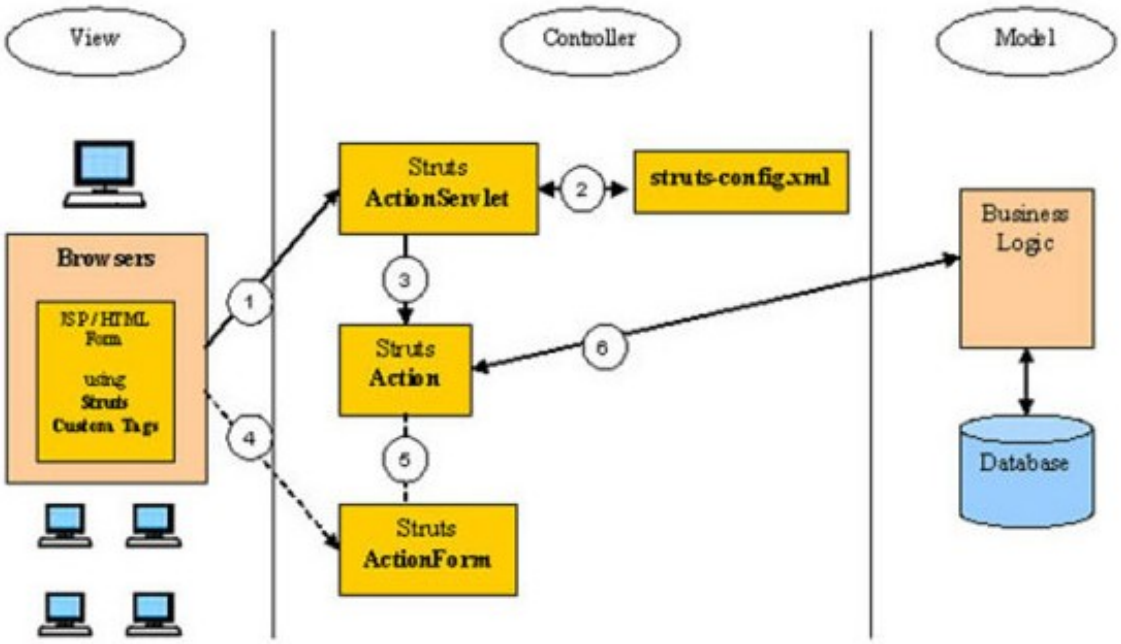


Figura 2 – Fluxo de uma aplicação que utiliza *Struts*[3]

1. Cada solicitação HTTP tem que ser respondida neste mesmo protocolo. Esta solicitação normalmente é definida como *requisicao.do* (nome lógico para a requisição do usuário);
2. A solicitação *requisicao.do* é mapeada no arquivo *struts-config.xml*, no qual estão todas as definições do componente *controller* do *Struts*. Este arquivo é lido por um *ActionServlet* (que fará efetivamente o papel do *controller* da aplicação) na inicialização da aplicação, criando um banco de objetos com o arquivo de configuração. Neste arquivo, são definidos os *Actions* (requisições dos usuários) para cada solicitação;
3. O *ActionServlet* define o *Action* correspondente para a solicitação. Um *Action* pode validar a entrada de dados e acessar o *model* para recuperar informações nos bancos de dados e outros serviços de dados;
4. A requisição HTTP pode ser feita também através de um formulário HTML (*form*). Em vez de fazer com que cada *Action* retire os valores do campo da solicitação, o

[3] Fonte: http://www.linhadecodigo.com.br/Artigo.aspx?id=1045&pag=1.

ActionServlet coloca a entrada em um *JavaBean*. Estes *JavaBeans* são definidos como *FormBeans* no *Struts* e estendem a classe *org.apache.struts.action.ActionForm*;
5. O *Action* pode acessar o *FormBean*, efetuar qualquer operação e armazenar o resultado em um *ResultBean*;
6. O *Action* interage com o *model* onde uma base de dados poderá ser atualizada.

Em geral, o *Struts* não apresenta a resposta em si, mas envia a solicitação para outro recurso, como uma página *JSP*. O *Struts* fornece a classe *ActionForward* para armazenar o caminho para uma página sob um nome lógico. Desta forma, o endereço ficará oculto para o usuário, de maneira que seja visualizado apenas o nome definido para o caminho (e.g., *resposta.do*). Esse recurso evita que o usuário possa visualizar uma versão desatualizada da aplicação, já que as requisições serão feitas apenas para nomes lógicos. Ao completar a lógica de negócio, o *Action* selecionará e retornará um *ActionForward* para o *servlet* e, então, este utilizará o caminho armazenado no objeto *ActionForward* para chamar a página e completar a resposta para o *view*. Tais detalhes da aplicação são definidos no objeto *ActionMapping*, onde cada *ActionMapping* está relacionado a um caminho específico. Quando este caminho for selecionado, como o caso de *requisicao.do*, o *servlet* irá recuperar o objeto *ActionMapping*. O mapeamento informará ao *servlet* quais *Actions*, *ActionForms* e *ActionForwards* utilizar.

Questão 6 [1 ponto]

Explique o que é Arquitetura de Referência e descreva quais são os seus elementos constituintes. Utilize um exemplo para ilustrar sua resposta.

Resposta:

Uma arquitetura de referência é um padrão de arquitetura que descreve todos os sistemas num domínio. Se uma arquitetura pode ser vista como um projeto reutilizável, então uma arquitetura de referência constitui um projeto reutilizável para uma classe de sistemas em um domínio específico. Dessa forma, uma arquitetura de referência compreende os seguintes elementos: (i) *estilo arquitetural*: serve para definir os tipos de componentes e conectores, bem como permite a configuração de uma aplicação (note que uma arquitetura de referência pode combinar estilos arquiteturais); (ii) *componentes típicos*: envolve os componentes típicos e variantes deles encontrados no estilo; e (iii) *interface com outros domínios*: necessidade de descrever a interface com outros domínios (isso pode ocorrer se uma aplicação requer que algum serviço ou facilidade seja fornecido por algum outro domínio).

Como exemplos, podemos citar: (i) o próprio MVC, uma arquitetura de referência relativa o modelo de referência[4] GUI (*Graphical User Interface*), cuja arquitetura de software poderia ser, por exemplo, SWING ou *Struts* e JSF (*JavaServer Faces*); (ii)

[4] Um modelo de referência é uma divisão de funcionalidades juntamente com o fluxo de dados entre as partes. É uma decomposição padrão de um problema conhecido em partes que cooperativamente resolvem o problema. Advindos da experiência, os modelos de referência são características de um domínio maduro e descrevem em termos gerais como as partes se inter-relacionam para alcançar seu propósito coletivo.

Máquina Virtual, uma arquitetura de referência para o modelo de referência interpretadores, cuja arquitetura de software poderia ser, por exemplo, a JVM (*Java Virtual Machine*); e (iii) CORBA (*Common Object Request Broker Architecture*), uma arquitetura de referência para o modelo de referência sistemas distribuídos, cuja arquitetura de software poderia ser, por exemplo, MICO. CORBA é a arquitetura padrão criada pelo OMG (*Object Management Group*) para estabelecer e simplificar a troca de dados entre sistemas distribuídos heterogêneos. Em face da diversidade de hardware e software que encontramos atualmente, CORBA atua de modo que os objetos (componentes dos produtos de software) possam se comunicar de forma transparente ao usuário, mesmo que para isso seja necessário interoperar com outro software, em outro sistema operacional e em outra ferramenta de desenvolvimento. CORBA é um dos modelos mais populares de objetos distribuídos, juntamente com o DCOM (*Distributed Component Object Model*), formato proprietário da Microsoft, e EJB (*Enterprise JavaBeans*), formato definido pela inicialmente pela SUN (outras arquiteturas de referência para o modelo de referência sistemas distribuídos).

Por exemplo, um sistema comercial faz uso da arquitetura CORBA que acomoda tanto o *estilo arquitetural* de camadas quanto o de objetos, incluindo seus *componentes típicos* e *interface com outros domínios*. A Figura 3 mostra uma solicitação de um cliente sendo passada a uma implementação de objeto na arquitetura CORBA. Nesse caso, tanto cliente quanto objeto se encontram em máquinas distintas e a comunicação ocorre via rede. Existe uma IDL (*Interface Definition Language*) entre objetos e a ORB (*Object Request Broker*). Perceba que tanto cliente quanto objeto são isolados da ORB por meio da IDL. Isso porque a arquitetura CORBA requer que toda interface de objeto seja expressa por meio de IDL. Note que os clientes vêem apenas a interface de um objeto. Isso assegura a substituição ou modificação de qualquer implementação por trás da interface. Dessa forma, poderia-se conectar ou desconectar componentes sem implicar em mudanças nos demais componentes. A ORB é encarregada de tratar as comunicações via rede para clientes e objetos. Perceba que qualquer solicitação de um cliente não passa diretamente para uma implementação de objeto. As solicitações são sempre tratadas pela ORB; nesse caso, qualquer invocação a um objeto é passada primeiro a ORB. A figura dá ênfase à comunicação de ORB para ORB. Entretanto, não há qualquer diferença seja na implementação do objeto local ou remoto. Se remota, a invocação passa da ORB do cliente para a ORB da implementação do objeto. É importante acrescentar que as IDLs provêem suporte à distribuição de várias formas: dão ênfase ao encapsulamento e definem operações e tipos não ambiguamente. Além disso, um repositório de interfaces ou *interface repository* (IR) compartilhado assegura que todas as ORBs na rede tenham acesso às definições de interface IDL. Cabe destacar que a arquitetura CORBA oferece suporte à manutenibilidade em que a adição, remoção ou modificação da implementação de objetos ocorre de forma independente, não acarretando necessidade de alteração em outros componentes. Ademais, ela provê suporte a confiabilidade e tolerância a falhas.

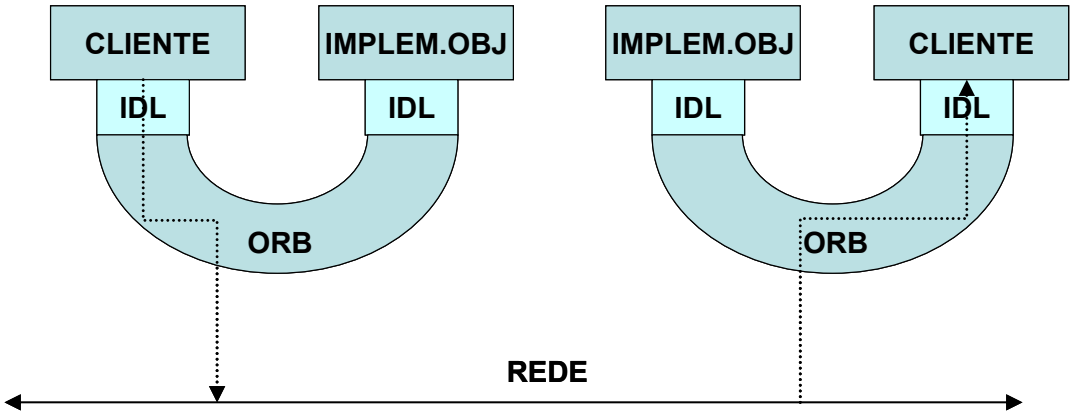


Figura 3 – Interação entre cliente e objeto na arquitetura CORBA[5]

[5] Fonte: *Arquitetura de Software*, Antonio Mendes, Editora Campus.