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 2° semestre de 2008.**

**Nome –**

---

Observações:

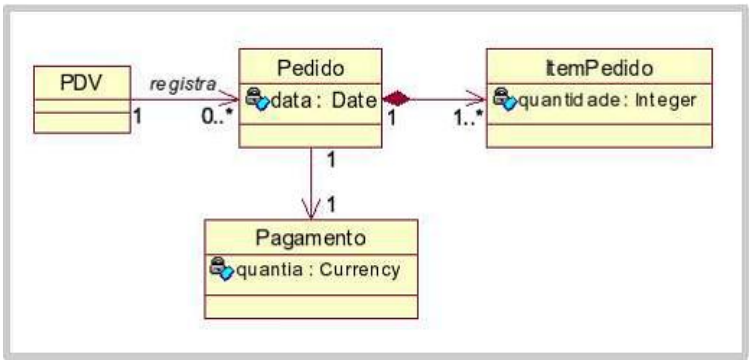
1. Prova com consulta.

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.

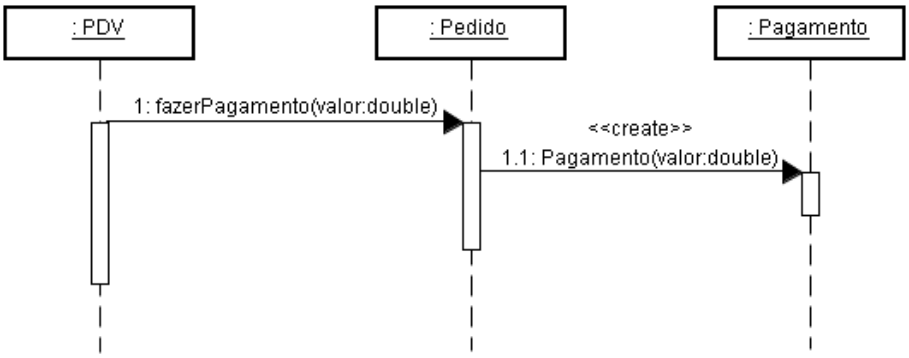
---

Questão 1 [1 ponto]

Desenvolva o modelo abaixo (através de modelo de seqüência) para determinar a responsabilidade de criação de pagamento, segundo o padrão *Creator*.

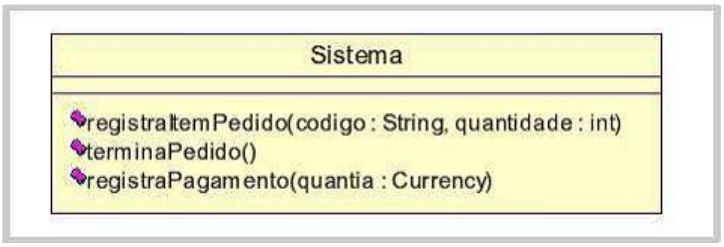



R: Conforme o padrão *Creator*, a responsabilidade de criação de uma instância da classe `Pagamento` deve ser atribuída à classe `Pedido`, uma vez que ela está intimamente relacionada com a classe `Pagamento` (um pagamento é referente a um pedido). Além disso, este modelo considera o padrão *Low Coupling*, uma vez que se evitou acoplar as classes `PDV` e `Pagamento`.



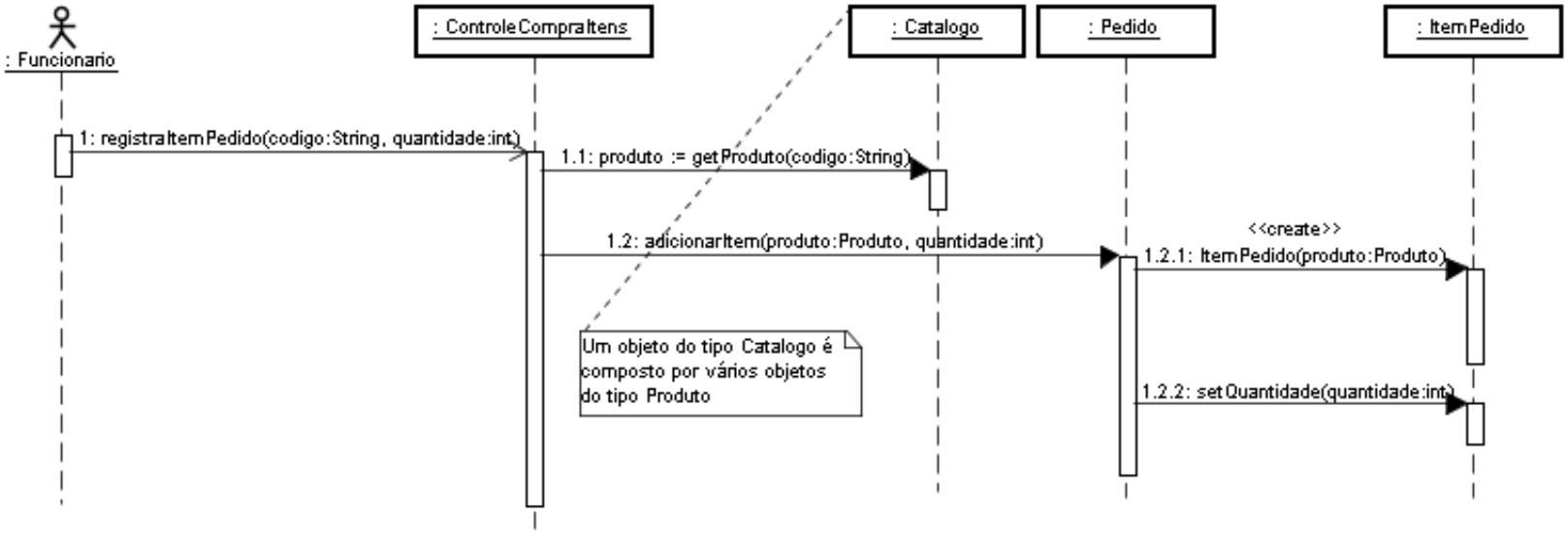

Questão 2 [1 ponto]

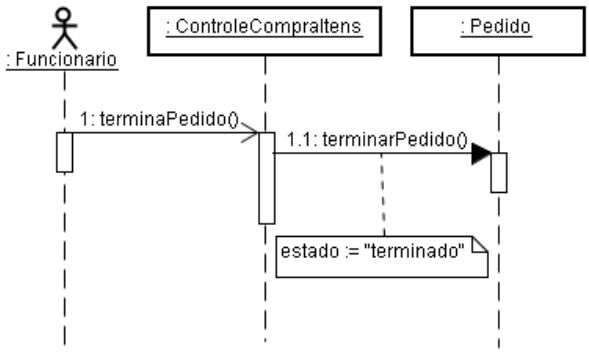
Para os eventos de sistemas listados abaixo, na classe virtual `Sistema` (vindos do caso de uso *Compra de itens*), defina uma atribuição de responsabilidades e argumente o motivo da sua escolha.

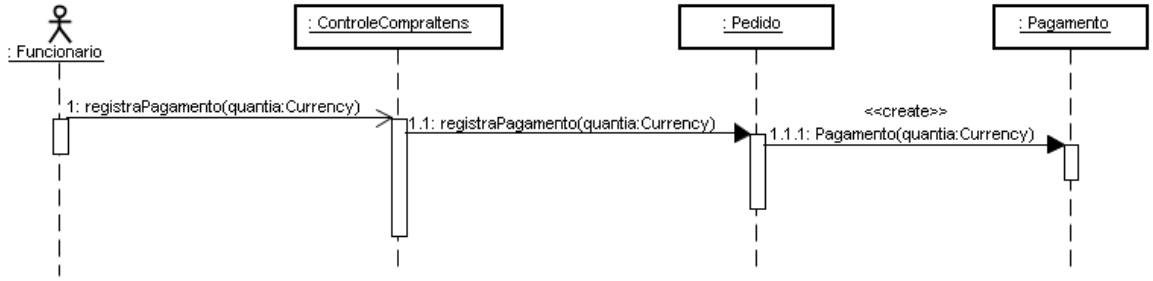



R: Sabe-se que os eventos de sistema estão associados às mensagens de sistema, que são geradas a partir dos passos dos casos de uso. Visando realizar uma atribuição de responsabilidades, ou seja, "quem" deve ser o responsável por tratar cada um desses eventos, utiliza-se o padrão *Controller*, desenvolvendo modelos de seqüência que explicitem essa atribuição. Nesses modelos, devem ser explicitadas quais classes recebem que requisições (provenientes das mensagens de sistema) e como essas classes devem estar estruturadas para tratá-las. No caso dos três eventos advindos do caso de uso *Compra de Itens* (apresentados pela classe virtual `Sistema`), iremos apresentar três modelos de seqüência que atribuem a um representante do caso de uso em questão (tratador artificial), a classe `ControleCompraItens`, a responsabilidade por tratar as mensagens relacionadas a esses eventos.

Como argumento geral para essa escolha, cita-se a separação *view-controller*, que facilita a reutilização de componentes específicos de negócio (lógica) diante de quaisquer alterações na interface com o usuário. Ou seja, delegar a responsabilidade de uma operação de sistema a um controlador apóia a reutilização da lógica em futuras aplicações e, como a lógica não está ligada à camada de interface, pode ser substituída por uma interface diferente.


: Funcionario
: ControleCompraItens
: Catalogo
: Pedido
: ItemPedido
1: registraItemPedido(codigo:String, quantidade:int)
1.1: produto := getProduto(codigo:String)
1.2: adicionarItem(produto:Produto, quantidade:int)
<<create>>
1.2.1: ItemPedido(produto:Produto)
1.2.2: setQuantidade(quantidade:int)
Um objeto do tipo Catalogo é composto por vários objetos do tipo Produto




Como argumento específico, relativo à escolha do controlador como o tratador artificial, ressalta-se que a utilização de uma única classe para todos os eventos de um caso de uso possibilita a manutenção de estado do caso de uso como atributos dessa classe. Os três eventos listados em `Sistema` constituem passos do caso de uso *Compra de Itens* e essa informação pode ser necessária para verificar se eles acontecem em uma seqüência válida ou no caso de se desejar ser capaz de raciocinar sobre o estado corrente de atividade e operações dentro do caso de uso em andamento. Por exemplo, pode ser necessário garantir que a operação *registraPagamento* não possa ocorrer até que a operação *terminaPedido* tenha ocorrido. Se esse for o caso, a informação de estado deverá ser buscada em algum lugar; o controlador é uma boa escolha, especialmente se o mesmo controlador for utilizado em todo o caso de uso (o que é recomendável). Além disso, a utilização desse controlador pode ser útil para não sobrecarregar nenhuma classe, pois, como cada caso de uso tem sua própria classe, mesmo que o sistema tivesse 1000 casos de uso, nenhuma classe ficaria muito grande.

Questão 3 [2 pontos]

Dê a sua avaliação sobre o acoplamento nos problemas abaixo e indique se é uma situação desejável.

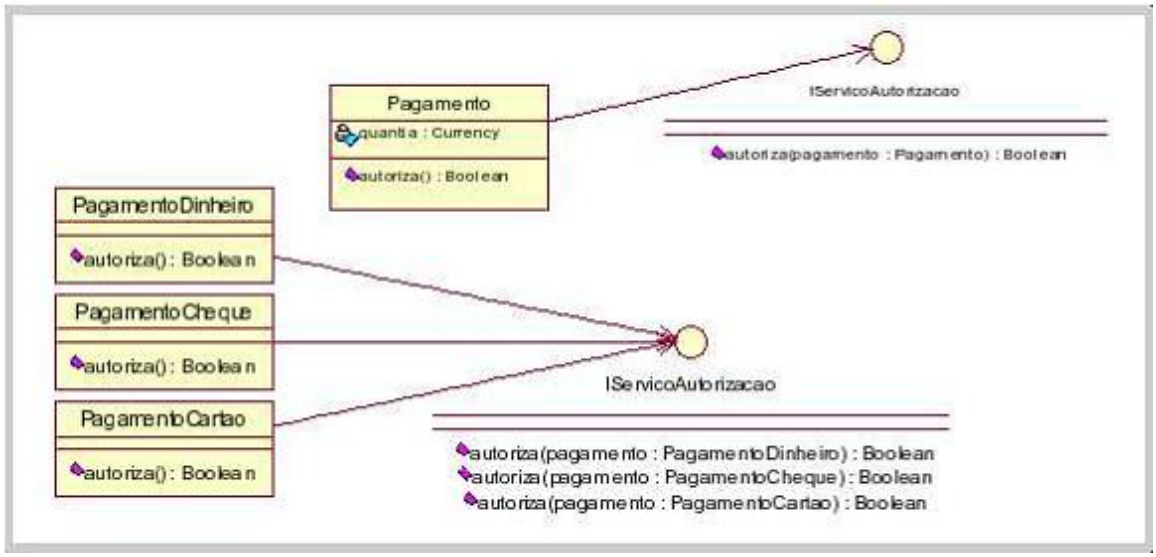
(a) [1 ponto] **CENÁRIO A**: um sistema onde todos os objetos herdam da classe `ObjetoPersistente`, para que seja possível implementar um mapeamento objeto-relacional automático;
(b) [1 ponto] **CENÁRIO B**: um sistema onde as classes têm somente atributos primitivos e métodos, sem nenhuma associação ou herança;

R: (a) Uma subclasse é fortemente acoplada à sua superclasse. Por isso, deve-se considerar cuidadosamente qualquer decisão de derivar de uma superclasse, pois é uma forma muito forte de acoplamento. O cenário apresentado é funcional, ou seja, não apresenta uma situação incorreta. No entanto, seguindo a idéia do padrão *Low Coupling*, a desvantagem desse uso de subclasses é que ele acopla fortemente objetos do domínio a um determinado serviço técnico e mistura diferentes interesses arquiteturais. Isso faz com que esses objetos estejam atrelados e dependam de um mecanismo de persistência específico. Em caso de descontinuidade do projeto desse mecanismo ou evolução não cuidadosa, ou ainda de manutenção corretiva, o sistema como um todo sofrerá seus reflexos (e.g., quando da necessidade de migração para outro mecanismo de persistência, esses objetos do domínio demandará adaptações, tornando-se oneroso para o projeto do sistema). Por outro lado, uma vantagem estaria na herança automática de comportamento de persistência, buscando facilitar o tratamento deste interesse. Enfim, esse cenário é representativo de um caso de alto acoplamento.

(b) Esse cenário representa o caso extremo de aplicação do padrão *Low Coupling*, no qual não há acoplamento entre classes. Isso não é desejável, porque contradiz uma metáfora central da tecnologia de objetos: um sistema é composto de objetos conectados que se *comunicam* por mensagens. Se esse padrão for aplicado em demasia, levará a um projeto ruim, com poucos objetos ativos, sem coesão, imprecisos e complexos, que fazem todo o serviço, e com muitos objetos passivos, com acoplamento zero, que agem como simples repositórios de dados. Ou seja, um grau de acoplamento entre classes é normal e necessário para criar um sistema orientado a objetos no qual as tarefas são executadas por uma colaboração entre objetos conectados. Dessa forma, este cenário não é funcional.

Questão 4 [1 ponto]

Qual das duas abordagens abaixo você considera melhor? Justifique, citando os prós e os contras de cada abordagem.



R: As abordagens apresentadas representam exemplos de aplicação do padrão *Pure Fabrication*. Considerando essas abordagens (a superior e a inferior), a abordagem superior pode ser considerada melhor. Um ponto positivo está no fato de que essa abordagem apresenta alta coesão, pois permite a construção de uma classe para cada serviço de autorização de pagamento, por meio da implementação da interface `IServicoAutorizacao`. Ou seja, para diferentes pagamentos (instanciados a partir de subclasses da classe `Pagamento` – seguindo o padrão *Polymorphism* –, tais como `PagamentoDinheiro`, `PagamentoCheque` e `PagamentoCartao`), existirão serviços distintos de autorização. Além disso, esse tipo de abordagem favorece à manutenção (inserção/alteração/remoção) de serviços de autorização de maneira mais organizada e sem grandes impactos no projeto do sistema. Por outro lado, existe um acoplamento direto entre a classe de negócio `Pagamento` e os serviços de autorização, tornando difícil a reutilização desses serviços; ainda, o número de classes do sistema será maior, pois existirão tantas classes a mais quantos forem os diferentes serviços de autorização.

Na abordagem inferior, um ponto negativo está no fato de que existe baixa coesão, uma vez que o serviço de autorização apresenta uma série de métodos específicos para cada pagamento, de forma que uma classe que implemente `IServicoAutorizacao` deverá implementar todos esses métodos ou atribuir código vazio àqueles não desejados. Fazendo-se uma analogia com os perigos detectados no projeto orientado a objetos com relação à herança, a situação mencionada conduz ao grave problema de falta de *comportamento fechado* em relação à interface, ou seja, nem todos os métodos da interface devem ser implementados efetivamente nas classes que a implementam. Além disso, essa abordagem se mostra mais difícil de manter, dado que cada novo serviço de autorização acarretará tanto na modificação da definição da interface quanto da(s) classe(s) que realiza(m) a sua implementação. Por outro lado, a classe que implementar `IServicoAutorizacao` encapsulará o serviço de autorização, utilizando-se de *overloading* para a seleção do método apropriado diante de cada chamada do método `autoriza()`, por algum pagamento distinto.

Questão 5 [2 pontos]

Considerando cada um dos estilos arquiteturais estudados (*Pipes & Filters*, em camadas, processos distribuídos e orientado a objetos), responda:

(a) [1 ponto] Enumere uma vantagem e uma desvantagem de cada um deles. Dica: Caso seja possível, consulte o capítulo 2 do livro do Mendes para enriquecer sua resposta.
(b) [1 ponto] Com base no resultado obtido no item anterior, sugira sistemas onde cada estilo arquitetural seja adequado. Em que situação a heterogeneidade de estilos pode ser necessária?

R: (a) *Pipes & Filters*

- Vantagens: simplicidade; e facilidades para compor e paralelizar o sistema;

- Desvantagens: baixo desempenho (abstração de dados primitiva e gerência de *buffers*); e difícil interatividade e cooperação entre filtros.

*Em camadas*
- Vantagens: flexibilidade; e reutilização de uma infra-estrutura de computação pré-existente;
- Desvantagens: determinação do número de camadas; e comprometimento do desempenho devido à comunicação entre as camadas.

*Processos distribuídos*
- Vantagem: decomposição do sistema em partes menores de modo a facilitar modificações necessárias;
- Desvantagens: restrições topológicas; sujeito a problemas de sincronização de mensagens.

*Orientado a objetos*
- Vantagens: permite a organização de programas; e ajuda a manter a integridade dos dados do sistema;
- Desvantagem: devido à granularidade fina dos objetos e a dependência entre eles, a manutenção e a reutilização podem ser dificultadas, pois para modificar ou reutilizar um objeto pode ser necessário conhecer e atuar sobre outros objetos do sistema.

(b) Como exemplos de sistemas para cada um dos estilos arquiteturais analisados, podemos citar: *pipes* & *filters*: compiladores; *em camadas*: sistemas *web*; *processos distribuídos*: sistemas para cálculo científico; e *orientado a objetos*: sistemas de informação. Apesar desses exemplos, em sistemas de grande porte, verifica-se que a arquitetura do sistema é na realidade uma combinação de estilos. Um exemplo em que a heterogeneidade de estilos pode ser adequada é a combinação do estilo arquitetural orientado para eventos e objetos na construção de jogos computacionais *on line*. Essa combinação visa permitir a decomposição de um sistema em termos de objetos (componentes de inferência, componente de interface gráfica e componente jogador computacional) que são mais independentes, além de possibilitar que atividades de computação e coordenação (de eventos) sejam realizadas separadamente. Há, ainda, a facilidade de reúso e manutenção, já que novos objetos podem ser facilmente adicionados. Essa característica, e interfaces bem definidas, facilitam ainda a integração. O mecanismo de invocação é não determinístico (isto é, ocorre de forma aleatória), uma vez que considera a recepção de eventos. Adicionalmente, os componentes têm seus dados preservados de qualquer modificação acidental, já que essas informações são encapsuladas em objetos, facilitando também a integração.

Questão 6 [1 ponto]

Discuta cada uma das DSSAs (*Domain Specific Software Architectures*) apresentadas no curso (*sistemas de informação*, *sistemas de tempo real*, *sistemas inteligentes adaptativos* e *sistemas para comunicação*) e os benefícios específicos de seu uso.

R: Arquiteturas em Sistemas de Informação correspondem às DSSAs que envolvem suporte a armazenamento, recuperação e processamento de dados, com o objetivo de prover o gerenciamento da informação. Como exemplo, pode-se citar arquiteturas que apóiam sistemas bancários, sistemas comerciais e sistemas de catalogação em bibliotecas, dentre outros. Nessa DSSA, dois conjuntos de atributos de qualidade devem ser levados em consideração: (i) desempenho, segurança, integridade e disponibilidade; e (ii) usabilidade e manutenibilidade. Uma das principais arquiteturas que suportam essa DSSA é a arquitetura cliente-servidor, que consiste no processamento distribuído entre dois elementos: *cliente*, que requisita serviços, e *servidor*, que realiza os serviços solicitados pelos clientes. Além disso, a arquitetura cliente-servidor exige a comunicação entre os dois elementos, necessitando, para isso, de uma rede entre os computadores e de um protocolo de comunicação. Um dos grandes benefícios dessa DSSA está na melhoria dos níveis de produtividade e na facilidade de tomada de decisão em uma organização, uma vez que permite aos funcionários maior acesso aos dados.

Arquiteturas em Sistemas de Tempo Real envolvem as DSSAs que devem apoiar respostas aos estímulos recebidos dentro de um estrito intervalo de tempo. Para isso, devem tratar a disponibilidade de componentes redundantes, a fim de prover suporte de tolerância a falhas e a necessidade de se fazer diagnósticos para a detecção de falhas e a reconfiguração do sistema com a substituição de componentes defeituosos. Dentre os benefícios agregados à utilização dessa DSSA estão o reúso de requisitos, de projeto, de componentes de software e de métodos, o que agrega, conseqüentemente, maior confiabilidade à classe de sistema de tempo real.

Arquiteturas em Sistemas Inteligentes Adaptativos se referem a DSSAs que devem atribuir maior atenção a características como habilidades de percepção, raciocínio e ação, uma vez que propiciarão a geração de sistemas inseridos em ambientes dinâmicos e complexos sob condições de incerteza e mudanças. Como requisitos funcionais, deve-se considerar: (i) percepção concorrente; (ii) sensibilidade a prioridades e a condições limites de tempo; (iii) controle global do comportamento; e (iv) raciocínio e ação. Um dos principais benefícios relacionados ao uso dessa DSSA contempla a idéia de maior modularidade, uma vez que o domínio dessa DSSA pode ser particionado em subdomínios envolvendo aplicações de robôs autônomos (que requer capacidade de navegação e planejamento de ações) e sistemas de monitoramento (exigindo capacidade de detecção de erros e classificação de padrões).

Arquiteturas em Sistemas para Comunicação é uma DSSA que foca na natureza distribuída dos sistemas atuais e deve suportar como mecanismo de comunicação o encapsulamento e a transparência de localização, com o uso da arquitetura em camadas. Nesse caso, as redes são estruturadas em camadas e protocolos, onde cada camada agrupa um conjunto de tarefas em um determinado nível de abstração e a quantidade e funcionalidade de cada camada varia de rede para rede. Como vantagens dessa DSSA,

encontram-se a redução da complexidade do sistema e a possibilidade de maior interoperabilidade.

Questão 7 [2 pontos]

Considerando as arquiteturas *web* responda:

a) [1 ponto] Quais os seus componentes fundamentais? Explique um canário típico de comunicação em um sistema *Web* que apresenta arquitetura em três camadas.
b) [1 ponto] Selecione dois estilos arquiteturais para *Web* e explicite suas características, vantagens e limitações.

R: (a) Os componentes fundamentais de arquiteturas *web* são: HTML (páginas *web* estáticas), componentes que rodam no cliente (e.g., Applets e ActiveX), scripts que rodam no cliente (e.g., JavaScript), scripts que rodam no servidor (e.g., PHP, ASP, JSP), componentes que rodam no servidor (e.g., Servlets e EJB) e banco de dados.

Um cenário típico de comunicação em um sistema *web* que apresenta arquitetura em três camadas poderia ser: inicialmente, o cliente no *browser* solicita um serviço ao servidor usualmente após o preenchimento de um formulário HTML; a seguir, o servidor interpreta a solicitação na camada de aplicação; se necessário, a camada de aplicação se comunica com a camada de armazenamento para acessar dados; assim, a camada de aplicação redireciona o fluxo para a camada de apresentação a fim de que o servidor construa uma página de resposta; por fim, o servidor retorna a página de resposta.

(b) *Thin Client*: representa um estilo fundamentado na mínima utilização dos recursos do cliente, exigindo mínima capacidade de seu navegador. Algumas de suas características são: (i) estilo mais apropriado para aplicações *Internet*; (ii) baixo controle da configuração do navegador cliente; e (iii) toda a lógica de negócio reside no servidor. Suas vantagens: (i) independência total de plataforma ou navegador; e (ii) concentração da lógica de negócio no servidor. Apesar disso, apresenta limitações: (i) interface com o usuário limitada pela linguagem HTML; e (ii) maior número de transações com o servidor.

AJAX: consiste em um estilo baseado na utilização de *scripts* de cliente de forma assíncrona, permitindo a carga parcial de páginas. Algumas de suas características são: (i) permite a carga parcial de dados em uma página e (ii) combina *JavaScript* e XML. Uma vantagem está na carga parcial e assíncrona de dados para o cliente, ao passo que uma limitação se refere à dependência (limitada) com o navegador.