GOVERNO DO ESTADO DO CEARÁ
Secretaria da Ciência, Tecnologia e Educação Superior

UNIVERSIDADE ESTADUAL VALE DO ACARAÚ

Engenharia de Software

NEaD UVA
Núcleo de Educação a Distância
Universidade Estadual Vale do Acaraú

ENGENHARIA DE SOFTWARE

MODULO I – COMPONENTES DE SOFTWARE

No final dos anos 90, projetistas dos meios empresarial e acadêmico comeram a observar que as tecnologias orientadas a objeto não eram suficientes para acompanha as rápidas mudanças de requisitos de sistemas de software. O desenvolvimento orientado a objeto não tinha conduzido a um extensivo reuso, como originalmente era esperado. Uma das razoes era a falta de produção de arquiteturas de software capazes de serem adaptadas aos requisitos que se modificavam> as metodologias orientadas a objeto não guiavam os projetistas a fazer uma clara separação entre os aspectos computacionais e os composicionais. Mesmo que as construções orientadas a objetos encorajassem o desenvolvimento de modelos que refletiam os objetos de domínio do problema.

Componentes de Software é o termo utilizado para descrever o elemento de software que encapsula uma série de funcionalidades. Um componente é uma unidade independente, que pode ser utilizado com outros componentes para formar um sistema mais complexo. Em programação orientada a objetos um componente é a classe que implementa uma interface e é autônomo em relação a outros componentes do sistema. Um sistema de software pode ser formado inteiramente somente por componentes, pois estes se interligam através de suas interfaces. Este processo de comunicação entre componentes é denominado composição.

Mas a definição é levada ainda além. Componentes são definidos para oferecer um certo nível de serviço. No caso dos componentes “comerciais de prateleira” (ou commercial off-the-shelf - COTS), o engenheiro de software sabe pouco ou nada sobre o funcionamento interno de um componente. Ao invés disso, ao engenheiro de software é dada apenas uma interface externa bem-definida a partir da qual ele deve trabalhar. O nível de serviço é portanto crucial e precisa ser acurado se quiser que a integração do componente ao sistema de software seja bem sucedida. Brown e Wallnau descrevem um componente de software como “uma unidade de composição contratualmente especificada e somente com dependências contextuais explícitas”. Ao contrário de objetos em POO, os componentes são usualmente construídos a partir de muitos “objetos” de software (embora a construção não seja confinada a POO) e fornecem uma unidade de funcionalidade coerente. Os assim chamados “objetos” trabalham em conjunto para realizar uma tarefa específica em um dado

nível de serviço. Componentes podem ser caracterizados com base em seu uso no processo de ESBC: Como mencionado acima temos os COTS. São componentes que podem ser comprados, pré-fabricados, com a desvantagem de que, no geral, não há código fonte disponível, e sendo assim, a definição do uso do componente dada pelo fabricante(desenvolvedor), e os serviços que este oferece, precisam ser confiavelmente testadas, como podem ou não ser acuradas. A desvantagem, entretanto, é que estes tipos de componentes deveriam(em teoria) ser mais robustos e adaptáveis, pois foram usados e testados(e reusados e re-testados) e muitas diferentes aplicações.

Conforme pesquisas realizadas, segundo Bosch [8], têm mostrado que raramente um componente é reutilizado como foi originalmente desenvolvido e que geralmente necessita de alguma forma de alteração para se adequar à arquitetura da aplicação ou aos demais componentes.

As abordagens mais comuns são:

- Encapsulamento caixa-branca(White box wrapping) – aqui, a implementação do componente é diretamente modificada para resolver incompatibilidades. Isso é, obviamente, possível apenas se o código-fonte do componente estiver disponível, algo extremamente improvável no caso de COTS.
- Encapsulamento caixa-cinza(Grey box wrapping) – Neste caso, a adaptação é feita por uso de uma biblioteca do componente fornecendo uma linguagem de extensão do componente ou API que possibilite a remoção ou mascaramento de conflitos.
- Encapsulamento caixa-preta(Black box wrapping) – Caso mais comum, onde não é possível o acesso ao código-fonte, e a única maneira de adaptar o componente é por pré/pós processamento a nível de interface.

É responsabilidade do engenheiro de software determinar se os esforços para encapsular um componente adequadamente são justificados, se seria menos custoso criar um componente que solucione os conflitos. Também, uma vez que um componente foi adaptado é necessário verificar a compatibilidade para integração e realizar-se testes para tentar antecipar quais comportamentos inesperados que surjam devido as modificações realizadas.

O encapsulamento/empacotamento consiste em produzir uma visão externa para um componente, isto é, uma interface, diferente de sua interface original com vista a adaptá-lo a requisitos específicos.

Principais finalidades:

- Incompatibilidade de interfaces (componentes que podem interagir, porém as interfaces não são compatíveis). Ocorre devido a assinaturas de métodos diferentes ou por heterogeneidade dos componentes(diferença na Linguagem de programação, na plataforma de execução e localização física);
- Alterar a funcionalidade original do componente. Isso é, incluir novas funcionalidades, como também alterar o tratamento original a determinadas invocações de métodos.
    - A colagem de componentes (glueing) trata o mesmo problema do empacotamento, isto é, viabilizar a operação conjunta de componentes originalmente incompatíveis. A diferença neste caso é que o tratamento dado ao problema é a inclusão de um novo elemento, a cola (glue), entre os componentes incompatíveis, possibilitando sua operação conjunta.
    - O elemento cola nada mais é que um terceiro componente, cuja interface possibilita sua conexão aos componentes Componente um e Componente dois e cuja funcionalidade consiste em compatibilizar a operação conjunta destes componentes. Seja a situação de colagem representada e considere-se que o Componente um seja implementado em Smalltalk e Componente dois, em Java. Com isto, o componente cola deve solucionar a heterogeneidade entre os outros dois componentes (que também pode envolver localização física e plataforma de execução), bem como eventuais incompatibilidades sintáticas e funcionais. Suponha-se que a questão da heterogeneidade seja resolvida com o uso de CORBA. Neste caso o componente cola mascararia o tratamento da heterogeneidade. A figura ilustra uma visão mais refinada do componente cola, considerando esta situação.
    - A cola é uma prática totalmente transparente para o usuário, uma vez que ele não sabe que três tecnologias diferentes foram utilizadas.

MODULO II – PROJETO DE COMPONETES BASEADOS EM CLASSES

As métricas de coesão e acoplamento tratam de avaliar como os componentes dependem uns dos outros. Neste caso, não são considerados como os requisitos funcionais, que levaram à criação destes componentes, são dependentes uns dos outros e nem se o conjunto de requisitos funcionais escolhido é apropriado. Quando se escolhe um conjunto de requisitos funcionais muito interdependentes, os componentes da solução gerada tendem a ser interdependentes também.

O desenvolvimento de software orientado a objetos tem como um dos seus preceitos aumentar a coesão e diminuir o acoplamento entre os módulos do sistema. O desenvolvimento orientado a objetos facilita para o desenvolvedor criar componentes mais reutilizáveis. Para isso, a orientação a objetos disponibiliza abstrações como classes, objetos, interfaces, atributos e métodos. Além disso, a orientação a objetos introduziu conceitos como encapsulamento, herança e polimorfismo para aumentar a reutilização e a extensibilidade e facilitar a manutenção de sistemas de software.

Coesão

Coesão pode ser descrita como "exclusividade de enfoque" de um componente. No contexto de projeto no nível de componente para sistemas orientados a objetos, coesão implica que um componente ou classe encapsule os atributos e operações muito relacionados entre si e com a classe ou componente propriamente dito.

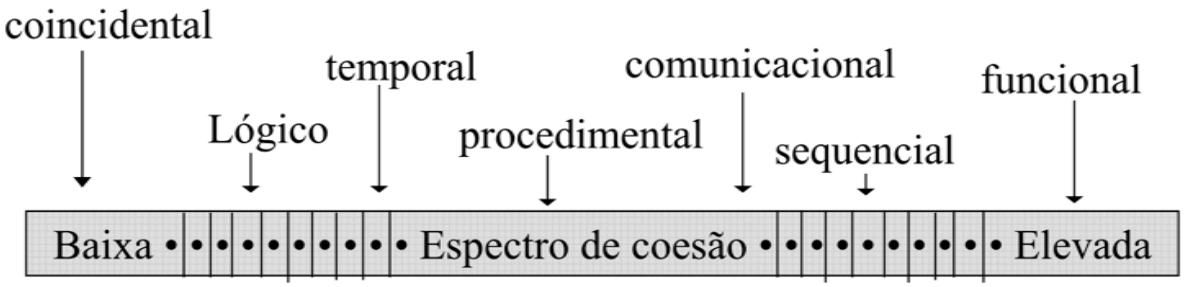
"Coesão é uma medida da força funcional relativa de um módulo".

Em outras palavras a coesão mede o grau com que as tarefas executadas por um único módulo se relacionam entre si. Para software orientado a objetos, coesão também pode ser conceituada como sendo o quanto uma classe encapsula atributos e operações que estão fortemente relacionados uns com os outros.
Uma classe com baixa coesão faz muitas coisas não relacionadas e leva aos seguintes problemas:

- Difícil de entender
- Difícil de reusar
- Difícil de manter
- Problemas com mudanças

Tipos de Coesão:

- Coincidente (pior)
- Lógico
- Temporal
- Procedural
- De comunicação
- Sequencial
- Funcional (melhor)



Coesão coincidente

- Há nenhuma (ou pouca) relação construtiva entre os elementos de um módulo
- No linguajar OO: Um objeto não representa nenhum conceito OO, Uma coleção de código comumente usado e herdado através de herança(provavelmente múltipla).

Coesão lógica

- Um módulo faz um conjunto de funções relacionadas, uma das quais é escolhida através de um parâmetro ao chamar o módulo.
- Semelhante a acoplamento de controle

Coesão temporal

- Elementos estão agrupados no mesmo módulo porque são processados no mesmo intervalo de tempo
- Exemplos Comuns:
  - Método de inicialização que provê valores defaults para um monte de coisas diferentes
  - Método de finalização que limpa as coisas antes de terminar

Coesão procedural

- Associa elementos de acordo com seus relacionamentos procedurais ou algorítmicos.
- Um módulo procedural depende muito da aplicação sendo tratada.

- Junto com a aplicação, o módulo parece razoável.
- Sem este contexto, o módulo parece estranho e muito difícil de entender.

Coesão de comunicação

- Todas as operações de um módulo operam no mesmo conjunto de dados e/ou produz o mesmo tipo de dado de saída.
- Cura: isole cada elemento num módulo separado.
- "Não deveria" ocorrer em sistemas OO usando polimorfismo (classes diferentes para fazer tratamentos diferentes nos dados) .

Coesão sequencial

- A saída de um elemento de um módulo serve de entrada para o próximo Elemento.
- Cura: decompor em módulos menores.

Coesão funcional (a melhor)

- Um módulo tem coesão funcional se as operações do módulo puderem ser Descritas numa única frase de forma coerente.

- Num sistema OO
  - Cada operação na interface pública do objeto deve ser funcionalmente coesa.
  - Cada objeto deve representar um único conceito coeso.

Acoplamento

"Acoplamento é uma medida da interdependência relativa entre os módulos". Para sistemas de software orientados a objetos pode-se definir acoplamento como sendo o grau com o qual classes estão conectadas entre si. Existem métricas definidas na literatura para coesão de um módulo e acoplamento entre módulos.

Acoplamento de classes pode manifestar-se de uma variedade de modos. Lethbridge e Laganiére[LET01] definem as seguintes categorias de acoplamento:

Acoplamento por conteúdo.

Ocorre quando um componente "sub-repticiamente modifica dados internos a outro componente" [LET01]. Isso viola ocultação de informação – um conceito básico de projetos.

Acoplamento comum.

Ocorre quando certo número de componentes faz uso de uma variável global. Isso seja algumas vezes necessário (por exemplo, para estabelecimento de valores padrão que são aplicáveis ao longo de toda aplicação), acoplamento comum pode levar à propagação descontrolada de erros e efeitos colaterais imprevisíveis quando modificações são feitas.

Acoplamento por controle.

Ocorre quando operaçãoA() invoca operaçãoB() e passa um sinal de controle para B. O sinal de controle então "dirige" o fluxo lógico dentro de B, O problema com essa forma de acoplamento é que uma modificação não relacionada em B pode resultar na necessidade de modificar o significado do sinal de controle que A passa. Se isso for ignorado, resultará em erro.

Acoplamento carimbado.

- Ocorre quando a ClasseB é declarada como um tipo de argumento de um operação da ClasseA. Como ClasseB é agora uma parte da definição da ClasseA, modificar o sistema torna-se mais complexo.

Acoplamento por dados.

- Ocorre quando operações passam longas cadeias como argumentos de dados. A "largura de banda" de comunicação entre classes e componentes aumenta e a complexidade da interface aumenta. Teste e manutenção são mais difíceis.

Acoplamento por chamada de rotina.

- Ocorre quando uma operação chama outra. Esse nível de acoplamento é comum e é frequentemente necessário. No entanto, não aumenta a conectividade de um sistema.

Acoplamento por uso de tipo.

- Ocorre quando um componente A usa um tipo de dado definido em um componente B (isso ocorre sempre que "uma classe declara uma instância de variável ou variável local como tendo outra classe para seu tipo" [LET01]). Se a definição de tipo muda, todo componente que usa a deinição deve modificar-se também.

Acoplamento por importação ou inclusão.

- Ocorre quando o componente A importa ou inclui um pacote ou conteúdo do componente B.

Acoplamento externo.

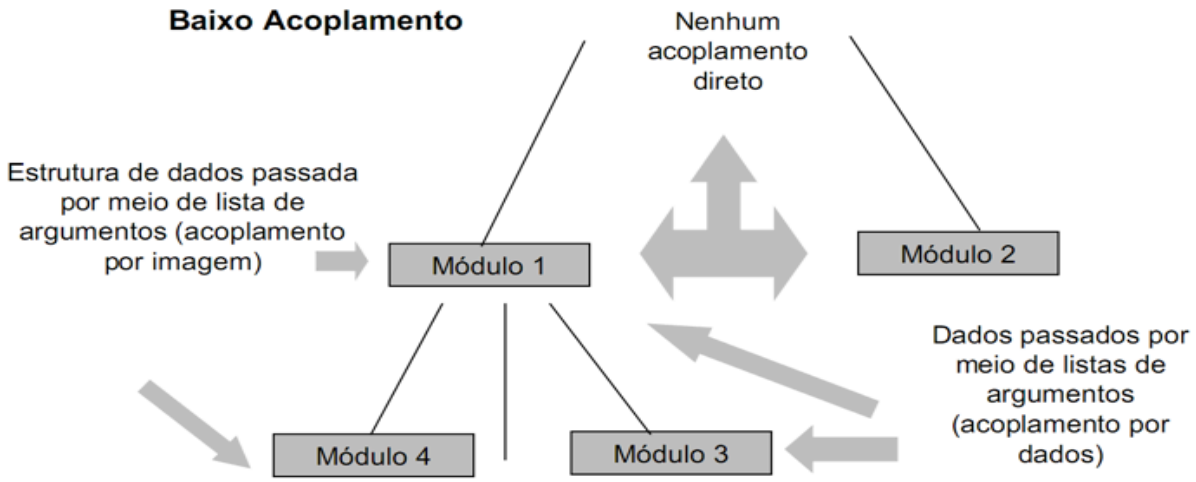
- Ocorre quando um componente comunica ou colabora com componentes de infra-estrutura (funções de sistema operacional, facilidades de bancos de dados, funções de telecomunicações). Embora esse tipo de acoplamento seja necessário, deve ser limitado a um pequeno número de componentes ou classes dentro de um sistema.

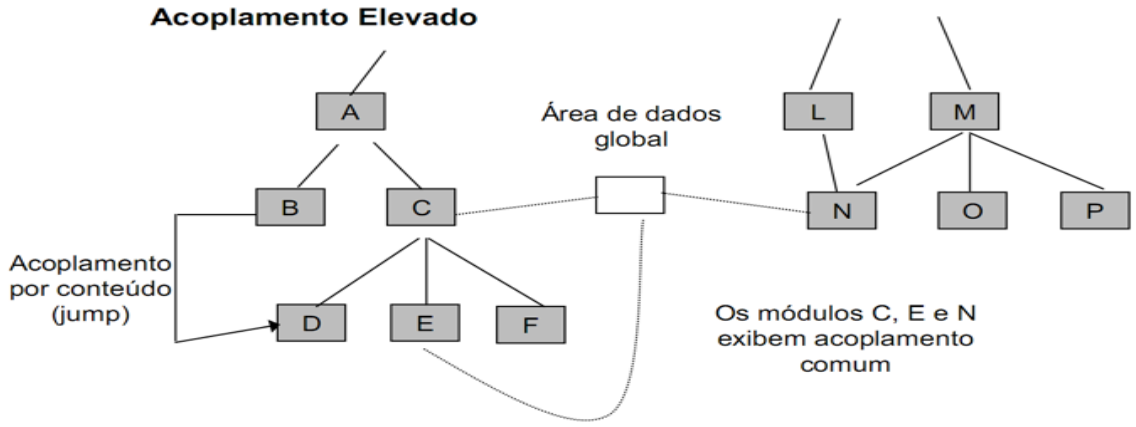
O software deve se comunicar interna e externamente. Assim, acoplamento é um fato da vida. No entanto, o projetista deve trabalhar para reduzi-lo sempre que possível e entender as ramificações do alto acoplamento quando ele não pode ser evitado.

Com uma classe possuindo forte acoplamento, temos os seguintes problemas:

» Mudanças em uma classe relacionada força mudanças locais à classe
» A classe é mais difícil de entender isoladamente
» A classe é mais difícil de ser reusada, já que depende da presença de outras classes.

Exemplos de acoplamento baixo e elevado:





MODULO III – CONDUÇÃO DO PROJETO DE NIVEL DE COMPONENTES

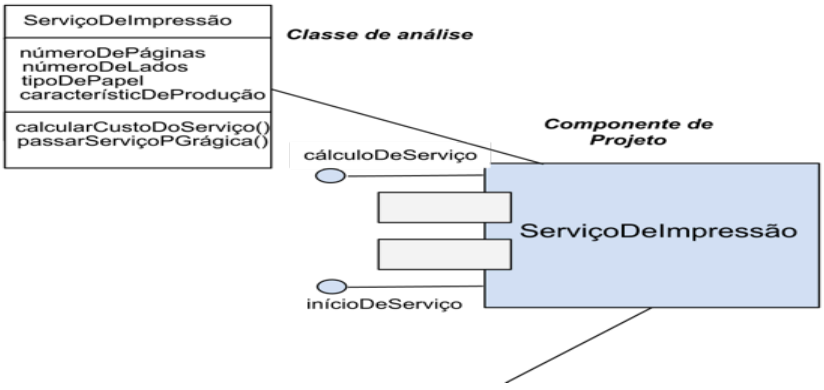
Após ser obtidas as informações dos modelos de análise e arquitetural, o projetista deve transformar estas informações em uma representação de projeto que sirva de base durante todo o processo de construção do sistema (codificação e teste).

Os seguintes passos representam um conjunto de tarefas típicas para o projeto no nível de componente, quando se é utilizado a um sistema que se utiliza do paradigma de orientação a objetos.

Passo 1: Identificar todas as classes de projeto que correspondam ao domínio do problema.

Usando os modelos de análise e arquitetural, cada classe de análise e componente arquitetural é elaborado como foi proposto anteriormente neste curso à projetos a nível de componente orientado a objetos. Como na visão orientada a objetos cada componente contém um conjunto de classes colaborativas, então deve-se elaborar todas as classes de análise que correspondam
ao domínio do problema.

Abaixo um exemplo de classe de análise, denominada ServiçoDeImpressão.



Passo 2: Identificar todas as classes de projeto que correspondam ao domínio de infra-estrutura.

Essas classes não são elaboradas no modelo de análise, mas devem ser descritas neste passo. Classes e componentes de infra-estrutura normalmente são componentes de IGU(Interface Gráfica de Usuário), componentes de sistema operacionais, componentes de gestão de objetos e dados e outros.

Exemplos: classes que fazem chamadas ao sistema operacional para determinar função, classes de acessos a banco de dados, etc...

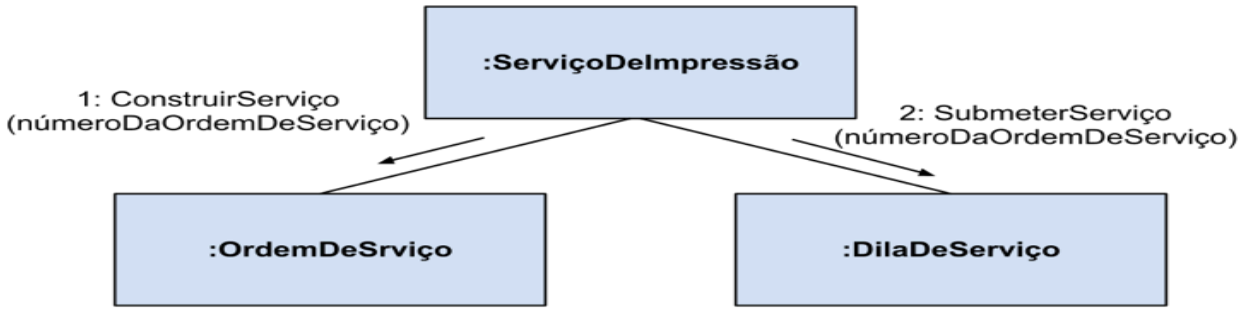
Passo 3: Elaborar todas as classes de projeto que não são adquiridas como componentes reusáveis.

Este passo requer que todas as interfaces, atributos e operações necessárias para implementar a classe sejam descritas em detalhes, ou seja, criar as classes que não focam o domínio do problema, nem classes de infra-estrutura. Heurísticas (formas de solucionar um problema) de projetos com coesão e acoplamento de componentes devem ser levados em conta.

Passo 3a. Especificar detalhes de mensagens quando classes ou componentes colaboram.

Como o modelo de análise faz uso de um diagrama para demonstrar as mensagens que as classes e componentes utilizam para se comunicarem, é necessário à medida que o projeto no nível de componente prossegue mostrar os detalhes dessas mensagens, ou seja, mostrar os detalhes das colaborações entre os objetos de um sistema.

A figura abaixo ilustra um exemplo de diagrama de colaboração simples para um sistema de gráfica.



Mensagens são passadas como é ilustrado por setas da figura. Á medida que o projeto prossegue, cada mensagem é elaborada expandindo sua sintaxe da seguinte modo em OCL:
[condição de guarda] expressão de sequência (valor de retorno):= nome da mensagem (lista de argumentos)

Em que [condição de guarda] especifica qualquer conjunto de condições que devem ser satisfeitas antes que a mensagem seja enviada; expressão de sequência é um valor inteiro ou outro indicador de sequência, que indica a ordem sequêncial de envio das mensagens;(valor retorno) é nome da informação que será retornada pela operação; nome da mensagem identifica a operação que deve ser envocada e lista de argumentos é a lista de atributos passados para a operação.

Passo 3b: Identificar interfaces adequadas para cada componente.

Uma interface é o equivalente a uma classe abstrata que disponibiliza uma forma de as classes se conectarem através de suas operações. Todas as operações, ou seja, as formas de conexão em uma classe abstrata (interface) deve ser coesiva; isto é, deve exibir processamento que enfoca função ou subfunção limitadas, devem ter "exclusividade de enfoque".

Passo 3c: Elaborar atributos e definir tipos de dados e estruturas de dados necessários para implementá-los.

Muitas das vezes as estruturas e tipos de dados são implementados de acordo com a liguagem de programação que será utilizada.

Em UML define-se um tipo de dados de atributo por meio da seguinte sintaxe:

nome: tipo-expressão = valor inicial {cadeia de propriedade}

Ex: peso-TipoDoPapel = "A" {contém 1 de 4 valores – A, B, C ou D}

Em que nome se refere ao nome do atributo e tipo-expressão é o tipo de dados; valor inicial é o valor que o atributo assume quando o objeto é criado e cadeia de propriedade define uma propriedade ou conjunto de características do atributo.

Passo 3d: Descrever fluxo de processamento em cada operação em detalhe.

Deve ser descrito em detalhes toda a sequência algorítima das operações, isso pode ser realizado através da utilização de uma linguagem de programação baseada em pseudocódigo (por exemplo: Portugol) ou com um diagrama de atividade UML.

Cada componente é elaborado através de iterações que utilizam o conceito de refinamento passo a passo. Uma iteração deve e pode expandir o nome da operação. Por exemplo, a operação calcularCustoPapel() pode ser expandida para:

calcularCustoPapel(peso, tamanho, cor):numérico

Se o algoritmo para implementar certa função é simples não necessita de nenhuma elaboração adicional. Entretanto, se o algorítmo for mais complexo, mais elaboração é necessária.

Passo 4: Descrever fontes de dados persistentes (bancos de dados e arquivos) e identificar as classes necessárias para geri-los.

Em muitos casos esses mecanismos de armazenamento de dados persistentes são inicialmente especificados como parte do projeto arquitetural. No entanto, à medida que o projeto prossegue é útil especificar detalhes sobre a estrutura e organização desses fontes de dados persistentes.

Por exemplo pode se especificar qual a tecnologia de dados utilizados (banco de dados ou arquivos), caso seja banco de dados deve-se indicar qual o SGDB(Sistema Gerenciador de Banco de Dados) utilizado.

Além de especificar todas as classes que se utilizam e se relacionam com os dados mantidos pelo sistema.

Passo 5: Desenvolver e elaborar representações comportamentais para uma classe ou componente

Durante o projeto no nível de componente, é as vezes necessário representar o comportamento de uma classe de projeto. O comportamento dinâmico de um objeto, ou seja, uma instância de uma classe de projeto quando o programa é executado, é afetado por eventos externos a ele e pelo seu estado atual. Pode se entender melhor o comportamento de uma classe examninando todos os casos de uso em que tratam dessa classe de modo relevante. Esses casos de uso ajuda o projetista a conhecer melhor os eventos que afetam o objeto e os estados pelos quais o objeto passa durante toda a execução do sistema.

A transição de um estado para outro que ocorre como consequência de um

evento, pode ser representado na forma:

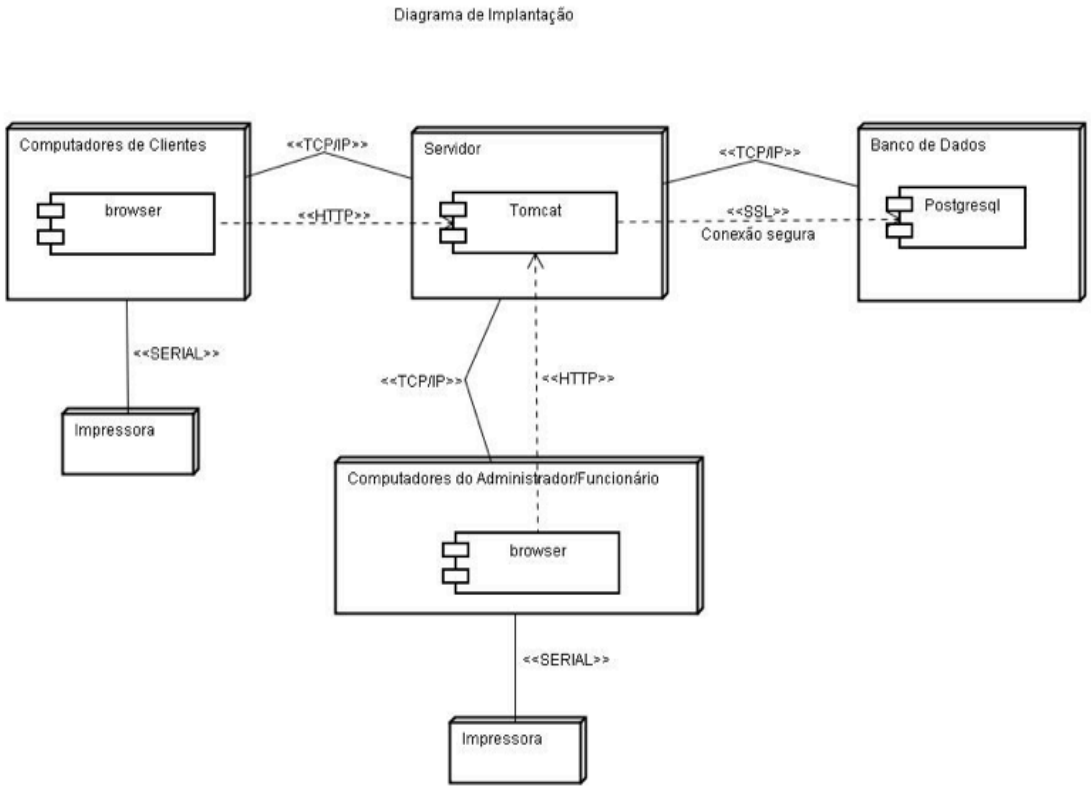
nome-evento(lista-parâmetros) [condição-de-guarda]/expressão de ação

Em que nome-evento identifica o nome; lista-parâmetros se refere aos dados associados ao evento; condição-de-guarda (especificada em OCL) especifica uma condição que deve ser satisfeita para que o evento ocorra e expressão de ação define a ação que ocorre quando a condição é verdadeira.

Passo 6: Elaborar diagramas de implantação para fornecer detalhe adicional de implementação.

Diagramas de implantação (como já vimos) são usados como parte do projeto arquiteural. As funções principais do sistema (algumas vezes representado como subsistemas) são representados de acordo com o ambiente específico de hardware e sistema operacional a ser usado e a localização de pacotes nesse ambiente é especificado.

Abaixo um exemplo de diagrama de implantação, em que mostra quais os tipos de SO, SGBD, dentre outras informações de um ambiente computacional de um projeto.



Passo 7: Fabricar toda a representação do projeto no nível de componente e sempre considerar alternativas.

Como o projeto pode ser é um processo iterativo. O primeiro modelo no nível de componente criado não será tão completo, consistente ou preciso quanto ao modelo que será criado quando o projeto já estiver em fase de desenvolvimento. É essencial recriar o modelo no nível de componente à medida que o projeto é implementado.

Além disso, o projetista não deve ter uma visão restrita e definitiva acerca do domínio do problema do sistema. Deve-se sempre haver outras soluções alternativas de projetos. Os melhores projetistas consideram todas as soluções possíveis antes de decidirem pelo modelo final de projeto.