ULBRA
CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS
COMUNIDADE EVANGÉLICA LUTERANA "SÃO PAULO"
Recredenciado pela Portaria Ministerial nº 3.607 - D.O.U. nº 202 de 20/10/2005

**Valdirene da Cruz Neves Júnior**

# MODELO E REPOSITÓRIO DE COMPONENTES PARA A *WEB*

**Palmas - TO**
**2013**

CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS
COMUNIDADE EVANGÉLICA LUTERANA "SÃO PAULO"
Recredenciado pela Portaria Ministerial nº 3.607 - D.O.U. nº 202 de 20/10/2005

**Valdirene da Cruz Neves Júnior**

# MODELO E REPOSITÓRIO DE COMPONENTES PARA A *WEB*

Trabalho de Conclusão de Curso (TCC) elaborado e apresentado como requisito parcial para obtenção do título de bacharel em Sistemas de Informação pelo Centro Universitário Luterano de Palmas (CEULP/ULBRA).

Orientador: Prof. M. Sc. Jackson Gomes de Souza.

**Palmas - TO**
**2013**

ULBRA CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS
COMUNIDADE EVANGÉLICA LUTERANA "SÃO PAULO"
Recredenciado pela Portaria Ministerial nº 3.607 - D.O.U. nº 202 de 20/10/2005

Valdirene da Cruz Neves Júnior

# MODELO E REPOSITÓRIO DE COMPONENTES PARA A WEB

Trabalho de Conclusão de Curso (TCC) elaborado e apresentado como requisito parcial para obtenção do título de bacharel em Sistemas de Informação pelo Centro Universitário Luterano de Palmas (CEULP/ULBRA).

Orientador: Prof. M. Sc. Jackson Gomes de Souza.

Aprovada em de 2013.

BANCA EXAMINADORA

_____

Prof. M. Sc. Jackson Gomes de Souza

Centro Universitário Luterano de Palmas

_____

Prof. M. Sc. Cristina D'Ornellas Filipakis

Centro Universitário Luterano de Palmas

_____

Prof. M. Sc. Edeilson Milhomem da Silva

Centro Universitário Luterano de Palmas

Palmas - TO
2013

Este trabalho é dedicado primeiramente a minha mãe, Maria da Penha, que me incentivou desde o início e é o principal motivo por eu chegar até aqui. A minha família pela fé e confiança demonstrada.

## AGRADECIMENTOS

Agradeço primeiramente a DEUS, por ter me iluminado durante essa caminhada. A minha família, pelo incentivo e colaboração, principalmente nos momentos difíceis. Agradeço também a todos os professores que me acompanharam durante a graduação, principalmente ao professor M. Sc. Jackson Gomes de Souza, que me orientou não somente na realização deste trabalho, mas na disciplina de estágio, no trabalho e até na vida. Aos meus amigos, que me apoiaram sempre. Aos meus colegas de classes, que às vezes unimos forças para estudar para as provas e fazer trabalhos. Agradeço a instituição, CEULP/ULBRA, por ser seu acadêmico e, principalmente, funcionário. Aos meus colegas que trabalharam comigo.

## RESUMO

Este trabalho apresenta conceitos referentes ao Reuso de *Software*, que vão desde objetos até *softwares* completos. Apresenta, também, conceitos sobre Engenharia de Domínio, que visa à busca e organização de componentes de *softwares* que serão utilizados em um projeto. Este trabalho apresenta, ainda, e destaca a Engenharia de *Software* Baseada em Componentes (CBSE), que visa à reutilização de componentes de forma padronizada. Na parte sobre CBSE são definidos e caracterizados componentes *softwares*, modelo de especificação de componentes e processo de desenvolvimento voltado para o reuso de componentes. Um componente de *software* é um pacote que pode ser implementado e instalado independente de um *software* maior. Já o modelo de especificação de componentes é um conjunto de regras que definem como os componentes serão implementados, documentado, disponibilizados, integrados e atualizados. Existem alguns modelos de especificação de componentes. Este trabalho apresenta dois deles, CORBA e WebService, realizado um comparativo entre os dois e cria um modelo de especificação de componentes voltado para o ambiente *web*. No modelo, são apresentadas as regras de especificação, o *middleware* e um repositório para armazenamento dos componentes. Como resultado deste trabalho, criou-se um modelo de especificação de componentes, que contém especificações para o desenvolvimento de componentes, *middleware* e um repositório de armazenamento de componentes.

**PALAVRAS-CHAVE:** Reuso de *Software*, Componentes, *Repono*, Repositório de Componentes.

# SUMÁRIO

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE LISTAGENS

# 1 INTRODUÇÃO

No contexto de empresas de desenvolvimento de *software*, geralmente ocorre de projetos de *softwares* serem construídos do início, sem a reutilização de partes de outro projeto, processo que, naturalmente, demanda gasto de tempo (na proporção da complexidade do projeto). Não é difícil perceber que *softwares* desenvolvidos em sequência, que compartilham funcionalidades, ou seja, que permitem reuso, possibilitam uma redução do custo de produção. Entretanto, esta atividade requer um processo gerenciado e sistemático, para que uma parte reutilizada de um sistema se integre a outra exigindo o mínimo de modificações. Este tipo de abordagem de reaproveitamento de código ficou conhecida como Engenharia de *Software* baseada em reuso, que é uma estratégia de Engenharia de *Software* na qual o processo de desenvolvimento é voltado para *softwares* existentes (SOMMERVILLE, 2007, p. 275).

A Engenharia de *Software* (ES) baseada em reuso varia desde a utilização de objetos e funções à reutilização do *software* na sua totalidade. Todavia, a tarefa de reutilização de componentes médios de um programa, que são partes de um sistema maior do que objetos ou funções, mas são menores que aplicações, não é trivial. A Engenharia de *Software* baseada em componentes é uma das técnicas de Reuso de *Software*. Esta abordagem visa à reutilização de componentes de *softwares* de forma padronizada, na qual uma aplicação, seguindo um modelo, consegue acoplar um componente implementando anteriormente sem a necessidade de modificação do mesmo para adaptação (SOMMERVILLE, 2007, p. 291).

Dentre as formas existentes de ES baseada em componentes podem ser citadas (D'SOUZA e WILLS, 1998, p. 403; SOMMERVILLE, 2007, p. 190):

- CORBA (*Common Object Request Broker Architecture*), em ambiente *desktop*; e
- WebServices, para reutilização por meio de serviços *web* (sobre o protocolo HTTP).

Os dois modelos, tanto o CORBA como o WebService, não permitem o desenvolvimento para a web em que os componentes possam ser executados no mesmo ambiente que a aplicação. No modelo de WebServices, por exemplo, a execução funciona em “caixa preta”, pois a aplicação cliente informa entradas à aplicação remota (o serviço) e recebe saídas, sem ter acesso a detalhes do processamento.

A existência de um modelo de especificação de componentes para a *web* e um repositório para armazenar esses componentes fará com que desenvolvedores, desse ambiente, não percam tempo implementando partes do sistema que já foram desenvolvidas antes, em outros sistemas, enfocando nas funcionalidades específicas da aplicação. Um exemplo de funcionalidade que pode ser reaproveitada é o esquema de paginação, que, normalmente, está presente em várias páginas de uma aplicação *web*.

A proposta desse trabalho é a criação de um modelo de especificação de componentes para a *web*. A criação, também, de um modelo de especificação de *middleware* e de repositório para esses componentes.

A ideia do desenvolvimento deste trabalho surgiu das dificuldades encontradas nas tentativas de reutilizar partes de sistemas *web*, já desenvolvidos, na Fábrica de *Software* do CEULP/ULBRA, pois os sistemas não seguem um modelo de implementação. A definição de um modelo de componentes de *software* tornará possível a reutilização desses componentes, fazendo que o desenvolvimento de aplicações torne-se ágil. O repositório concentrará os componentes desenvolvidos facilitando a localização e filtragem.

O presente texto está organizado da seguinte maneira: o referencial teórico é apresentado no capítulo 2, contendo os principais conceitos envolvidos no trabalho; posteriormente são apresentados os materiais e métodos utilizados no desenvolvimento do trabalho, no capítulo 3; o capítulo 4 contém os resultados e discussões, que apresenta o modelo e o repositório de componentes desenvolvimento; no capítulo 5 são feitas as considerações finais; por fim, no capítulo 6, as referências bibliográficas utilizadas no desenvolvimento do projeto.

# 2 REFERENCIAL TEÓRICO

Esta seção apresenta conceitos sobre Reuso de *Software*, Engenharia de *Software* baseada em componentes e Engenharia de Domínio, que servirão como base para criação do modelo de componentes e do repositório propostos nesse trabalho.

## 2.1 Reuso de *Software*

Segundo Sommerville (2007, p. 275) o Reuso de *Software* é a parte da Engenharia de *Software* que estuda como uma aplicação, ou parte dela, pode ser reutilizada na criação de um novo sistema:

> o processo de projeto, na maioria das disciplinas de engenharia, baseia-se no reuso de sistemas existentes ou componentes. Engenheiros mecânicos ou elétricos normalmente não especificam um projeto em que cada componente tenha de ser fabricado especialmente. Eles baseiam seus projetos em componentes já existentes e testados em outros sistemas. Esses componentes não são apenas pequenos componentes como flanges e válvulas, mas incluem subsistemas maiores, como motores, condensadores ou turbinas.

Para Pressman (1997, p. 729) a ideia de reutilização, dentro da Engenharia de *Software*, não era novidade. Desde o início da computação, ideias, objetos, abstrações e processos já eram reutilizados. O que vinha acontecendo de novidade, na época, era a necessidade de desenvolver sistemas de computadores muito complexos e de alta qualidade em um espaço de tempo curto.

D'Souza e Wills (1998, p. 454) definem reuso como uma variedade de técnicas destinadas a obter o máximo dos trabalhos que já foram implementados. O objetivo é não reinventar as mesmas ideias cada vez que for criado um novo projeto, mas sim capitalizar sobre esse trabalho e implantá-lo imediatamente em novos contextos. Dessa forma, pode-se oferecer mais produtos em menos tempo.

Um exemplo atual de reutilização de *software* em caixa branca é o *Twitter Boostrap*[1], um *framework* CSS, que permite reutilizar o modelo de interface do usuário no desenvolvimento de aplicações para a *web*. É possível reutilizar botões, ícones e outros elementos visuais. A Figura 1 apresenta um exemplo de utilização do *Twitter Bootstrap*:

[1] http://twitter.github.com/bootstrap/


**Figura 1: Exemplo de utilização do Twitter Bootstrap.**

A Figura 1 apresenta um formulário de autenticação de usuário utilizando o *Twitter Bootstrap*, no qual, a partir da utilização de classes CSS (*Cascading Style Sheet*) e do modelo HTML definidos pelo *framework*, a interface é estilizada. A classe CSS aplicada ao botão “Entrar” poderá ser utilizada em outros componentes, por exemplo, links, fazendo com que seja mantido o mesmo estilo visual.

Pelo exemplo da Figura 1 é possível entender que identificar os níveis de reuso de *software* é uma tarefa importante. Por exemplo, há o nível de reuso das classes CSS (que corresponde a reuso e padrão de interface) e há o nível de reuso que corresponde aos vários arquivos tanto com regras de estilo CSS quanto de comportamento e interação (em linguagem JavaScript). De acordo com Sommerville (2007, p. 276) as unidades de Reuso de *Software* variam de tamanhos:

- **reuso do sistema da aplicação:** uma aplicação inteira pode ser reutilizada, necessitando apenas da configuração para o novo ambiente. O reuso de aplicação pode, também, ser da junção de duas ou mais aplicações para gerar uma nova aplicação. A estrutura principal, ou núcleo, de uma aplicação pode ser reutilizado para criar outra aplicação;
- **reuso de componentes:** consiste no desenvolvimento de componentes de *software* padronizados, que seguem um modelo definido para que possam ser reutilizadas na composição de novas aplicações. Esse modelo definido

é um padrão para a implementação, documentação e implantação dos componentes. O Reuso de *Software* baseado em componentes será detalhado na seção 2.3;

- **reuso de objetos e funções:** é a reutilização de componentes, classes ou funções, que fazem uma única tarefa, por exemplo, a validação de um número de CPF.

Existem, também, diversas técnicas de reutilização de *software* que, de acordo com Sommerville (2007, p. 277), foram criadas ao longo dos últimos vinte anos e exploram as similaridades das aplicações de um mesmo domínio. A Figura 2 apresenta estas técnicas.

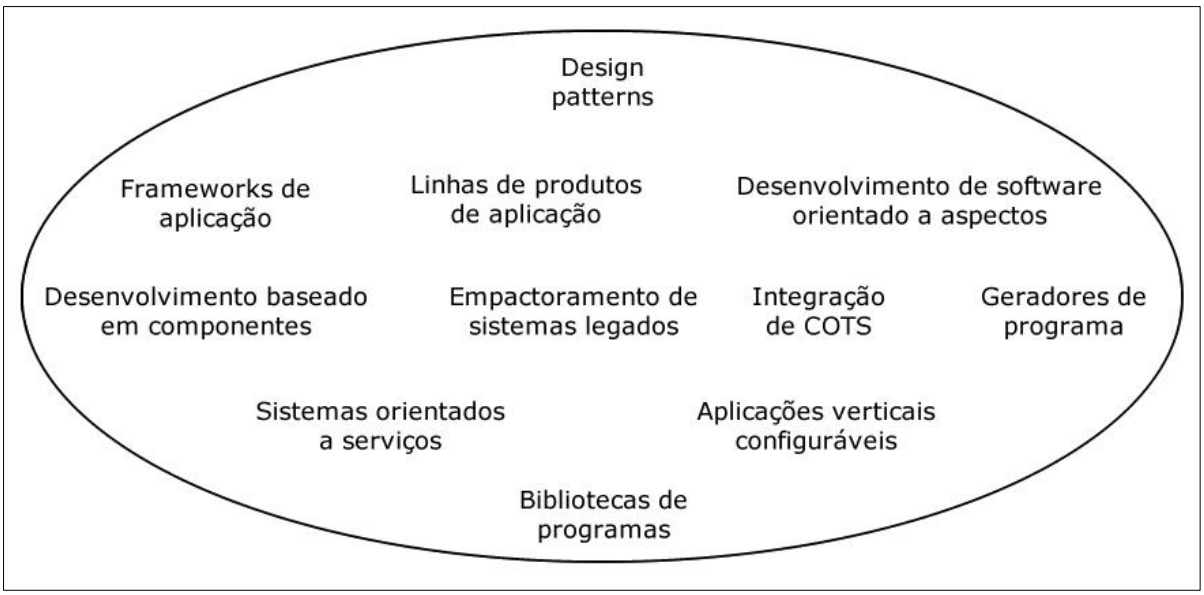


**Figura 2: Panorama de Reuso de *Software* (SOMMERVILLE, 2007, p. 277).**

Na Figura 2 são apresentadas as técnicas de reutilização de *software*, onde:

- ***design Patterns*** (Padrões de Projeto): na Engenharia de *Software*, representam a solução de um problema, não o código, mas a forma em que a solução será implementada, a estrutura das classes, os relacionamentos e as hierarquias (GAMMA et al., 2000, p. 19);
- **desenvolvimento baseado em componentes:** os componentes reusáveis não são desenvolvidos especificamente para uma aplicação. São

realizadas buscas por componentes que possam atender os requisitos, se não forem encontrados tais componentes, os mesmos serão implementados (SOMMERVILLE, 2007, p. 297);

- **linha de produtos de aplicação**: aplicações generalizadas, que podem ser vendidas para vários clientes sem a necessidade de modificação ou com poucas adaptações (SOMMERVILLE, 2007, p. 278);
- ***frameworks* de aplicação:** conjunto de classes para auxiliar na criação de aplicações;
- **integração de COTS:** componentes comerciais prontos para uso *(commercial off-the-self*), geralmente desenvolvidos com padrões bem conhecidos, como o CORBA;
- **geradores de programa:** *software* capaz de gerar aplicações ou fragmentos de aplicações para um domínio específico;
- **bibliotecas de programas:** classes e funções que implementam abstrações comumente utilizadas, como por exemplo, funções matemáticas;
- **desenvolvimento de *software* orientado a aspectos:** permite a separação dos componentes por ordem de importância para aplicação por meio de criação de módulos;
- **empacotamento de sistemas legados:** técnica de transformar sistemas antigos em pacotes para novos sistemas. Sistemas legados normalmente são grandes, implementados em linguagens não comuns atualmente, que torna o processo de empacotamento mais barato do que reimplementá-lo.

### *2.1.1 Vantagens e Desvantagens de Reuso de Software*

Para Sametinger (1997, p. 1) a reutilização de *software* apresenta diversas vantagens, entre elas pode-se destacar o tempo, que para o desenvolvimento de aplicações baseadas em partes de aplicações existentes, é menor. Isso influencia diretamente o preço do novo produto. Outro fator importante é a qualidade, uma vez que esses componentes desenvolvidos já foram testados e corrigidos.

Por outro lado, o Reuso de *Software* apresenta, também, algumas desvantagens, pois a manutenção em aplicações que utilizam componentes desenvolvidos por terceiros pode ser mais trabalhosa, principalmente se o código-

fonte desses componentes não estiver acessível. Sommerville (2007, p. 278) não recomenda o Reuso de *Software* para o desenvolvimento de aplicações que contêm o ciclo de vida muito longo, pois, provavelmente, irão aparecer novas funcionalidades e o sistema deve adaptar-se.

A seção 2.2 irá apresentar os conceitos relacionados à Engenharia de Domínio e as técnicas que poderão ser usadas para definição do que será reutilizado na construção de uma aplicação.

## 2.2 Engenharia de Domínio

Para Clements (1995 *apud* PRESSMAN, 2006, p. 666) a Engenharia de Domínio (ED):

> consiste em encontrar pontos comuns entre sistemas para identificar componentes que podem ser aplicados a muitos sistemas e identificar famílias de programas, que são posicionadas para tirar plena vantagem desses componentes.

Pressman (2006, p. 666), seguindo o mesmo pensamento, afirma que o objetivo da ED é identificar, construir, catalogar e disseminar os componentes de *software* que serão utilizados em um projeto. Vasconcelos (2007, p. 24) diz que na ED serão produzidos artefatos reutilizáveis que servirão de apoio na implementação do sistema em um determinado domínio. Sendo assim, entende-se que a ED visa identificar pontos comuns entre dois ou mais sistemas e saber o que pode ser reutilizado, ou seja, reuso de conhecimento do domínio. Este reuso é feito de forma sistemática.

A definição de um domínio específico para construção de um componente é muito importante, principalmente para organizações que têm uma cultura de reutilização, pois elas devem ter em mente a exigência do investimento realizado na construção desses componentes (D'SOUZA e WILLS, 1998, p. 456 e 457). Por exemplo, uma organização que desenvolve componentes para prevenção de colisão de aviões, não tem razões para fazer o trabalho extra de torná-los utilizáveis em navios, ou seja, se o produto lida apenas com o tráfego aéreo, não há razão para separar as peças da classe avião que poderiam se aplicar a navios ou a veículos em geral. Todas essas generalizações exigiriam ampliação das exigências do projeto, o que estaria além de suas necessidades imediatas.

A ED inclui algumas atividades, sendo, para Blois (2006, p. 11-13) e Vasconcelos (2007, p. 24-26), as três mais importantes a análise, o projeto e a implementação do domínio. As seções 2.2.1, 2.2.2 e 2.2.3 detalharão estas atividades.

### *2.2.1 Análise de Domínio*

De acordo Kang *et al*. (1990 *apud* BLOIS, 2006, p. 11) a Análise de Domínio (AD) "[...] busca identificar, coletar e organizar informações relevantes do domínio, utilizando para tal o conhecimento existente do domínio e métodos para a modelagem de informações". Essa ideia também é defendida por Pressman (2006, p. 666), quando indica que, na AD, são realizados os processos de: definição do domínio a ser analisado; categorização dos itens extraídos do domínio; coleta de amostras das aplicações desse domínio; análise de cada aplicação da amostra coletada e desenvolvimento do modelo de análise do projeto. A AD tem como objetivo maximizar a reutilização por meio da coleta de informações de maneira confiável e sistemática (OLIVEIRA, 2006, p. 7).

Dessa maneira, pode-se afirmar que o objetivo da análise de domínio, dentro da Engenharia de *Software*, é direcionar para um domínio mais específico para que as aplicações desse mesmo domínio possam partilhar características, o que facilita a reutilização de partes dessas aplicações. Essas características podem ser configuradas ou customizada.

Essa flexibilidade de configuração é chamada por Oliveira (2006, p. 7, 16, 17) de "variabilidades":

- **pontos de variação:** são os pontos em que começam as diferenças entre as características do domínio, ou seja, as variantes. Por exemplo, na utilização de um *framework* como reuso de *software*, podem existir variações do *framework* escolhido, como o SGBD utilizado para armazenar dados. A escolha dessa variação determina um ponto de variação;
- **variantes:** são as alternativas a partir de um ponto de variação. No exemplo do *framework*, a escolha do SGBD MySQL é uma variante, a escolha do Microsoft SQL Server é outra;

- **invariantes:** elementos fixos que não podem ser configuráveis. Por exemplo, alguns *frameworks* MVC determinam que o nome da classe que representa um controlador deve seguir o formato "[nome]Controller";
- **elementos opcionais:** elementos que não são obrigatórios no desenvolvimento da aplicação. No exemplo do *framework*, poderia ser um sistema *cache* de dados;
- **elementos mandatórios:** são elementos obrigatórios e devem estar presentes na aplicação. No exemplo do *framework*, poderia ser a plataforma (PHP, .NET etc.) em que ele foi implementado.

A fase de análise de domínio serve como base para criação de um projeto, a próxima seção apresenta sobre projeto de domínio.

### *2.2.2 Projeto do Domínio*

A fase de projeto de domínio tem por objetivo a definição de arquiteturas de *software* específicas de domínio (DSSAs – *Domain Specific Software Architectures*) com base nos requisitos e na análise de domínio (VASCONCELOS, 2007, p. 25). Para Blois (2006, p. 12) é nessa etapa que os modelos de projeto são construídos e, no desenvolvimento baseado em componentes, poderão ser modelos de componentes, classes, colaboração e sequência, referentes à estrutura interna dos componentes ou modelos de comunicação. As arquiteturas propostas nessa etapa são implementadas, como mostra a seção 2.2.3.

### *2.2.3 Implementação do Domínio*

Por fim, a implementação do domínio é a fase de codificação. Visa construir artefatos reutilizáveis, por meio de códigos fonte, com base na análise de domínio e as arquiteturas que foram propostas no projeto (BLOIS, 2006, p. 13).

A ED não é utilizada apenas para Engenharia de *Software* baseada em componentes, mas também para outras técnicas, como *frameworks* de aplicações

integração de COTS. A próxima seção aborda sobre Engenharia de *Software* baseada em componentes, que utiliza a ED.

## 2.3 Engenharia de *Software* Baseada em Componentes

De acordo com Sommerville (2007, p. 291) a Engenharia de *Software* baseada em componentes (CBSE – *Component-Based Software Engineering*) é uma das técnicas de reuso de *software* que visa à reutilização de componentes de *softwares* de forma padronizada. Ainda segundo o autor, uma aplicação, que segue um modelo, consegue acoplar um componente implementado anteriormente sem a necessidade de modificação do mesmo para adaptação. Por exemplo, um componente de máscara e validação de CPF, que segue o modelo de WebForms[2] da plataforma .NET, pode ser reutilizado no desenvolvimento de várias aplicações que utilizam o mesmo modelo.

Sommerville (2007, p. 292) apresenta alguns pontos essenciais para o desenvolvimento de *software* baseado em componentes:

- **componentes independentes:** permitem a separação entre a interface (do usuário) e a implementação, de tal forma que possa haver a troca do componente sem comprometer a aplicação;
- **padrões de componentes:** facilitam a integração dos componentes. Esses padrões devem estar definidos no modelo de componente (seção 2.3.1);
- ***middleware*:** fornece apoio para que os componentes se integrem, de forma que componentes independes trabalhem juntos ou dependentes se acoplem;
- **processo de desenvolvimento:** pois a utilização dos processos voltados para o desenvolvimento tradicional pode limitar a ES baseada em componentes.

Ainda segundo Sommerville (2007, p. 292), o desenvolvimento de *software* baseado em componentes encontra algumas dificuldades, como a confiabilidade, visto que alguns dos componentes desenvolvidos por terceiros não têm o código aberto, os quais podem conter códigos maliciosos. Outra dificuldade é a

[2] API da plataforma .NET para criação de páginas web.

certificação de componentes, que também está ligada à confiabilidade, pois a certificação serve para garantir quais componentes são confiáveis. Porém, essa certificação existe apenas em alguns contextos, como é o caso do *NuGet*[3], em que componentes passam por um processo de validação antes de serem disponibilizados.

A CBSE está diretamente ligada ao objetivo desse trabalho, que visa a um modelo de desenvolvimento de *software* baseado em componentes para *web*. A seção 2.3.1 irá detalhar sobre componentes de *software* e modelos de componentes de *software*.

### *2.3.1 Componentes e Modelos de Componentes*

Para Sametinger (1997, p. 2) os componentes são artefatos que identificam um *software,* os quais, apesar de seus detalhes de funcionamento estarem encapsulados, podem facilmente se combinar sem que um saiba detalhes internos um do outro. Neste mesmo sentido, D'Souza e Wills (1998, p. 387) compartilham da mesma ideia quando afirmam que componente de *software* é definido por como um pacote coerente de *software* que pode ser desenvolvido e instalado independentemente como uma unidade. Ainda segundo os autores, componente de *software* tem interfaces explícitas e bem definidas para os serviços que provê e pode ser utilizado para composição com outros componentes, sem alterações em sua implementação, podendo, eventualmente, permitir personalização de algumas de suas propriedades.

De acordo com Sommerville (2007, p. 295) os componentes são definidos por suas interfaces e, na maioria dos casos, podem assumir dois tipos: *requires* (requer) e *provides* (fornece), como mostrado na Figura 3.

[3] Gerenciador de pacotes para a plataforma .NET. Contém uma aplicação cliente e um repositório de pacotes central (NUGET, *online*): http://nuget.org/

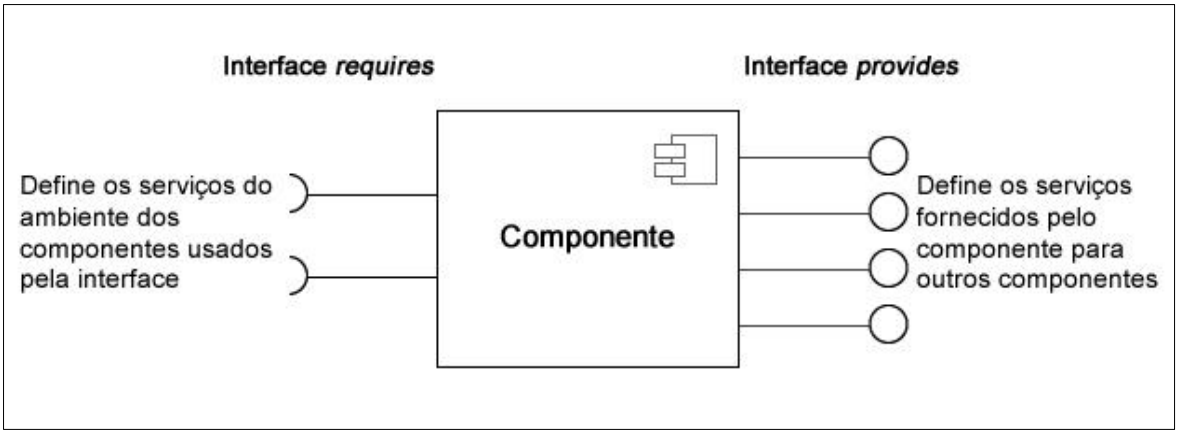


**Figura 3: Interfaces de componente (SOMMERVILLE, 2007, p. 294).**

A Figura 3 apresenta o modelo de diagrama UML 2 para representação de um componente, o qual contém as interfaces dos tipos *requires* e *provides*. As interfaces *requires*, à esquerda, representadas por um semicírculo, definem os serviços que devem ser oferecidos por outros componentes para o funcionamento deste, caso contrário, o componente não irá funcionar. As interfaces *provides*, à direita, representadas por um círculo, definem os serviços que são fornecidos pelo componente. Para exemplificar estes conceitos, a Figura 4 apresenta um diagrama UML 2 do componente hipotético *Player de mídia*.

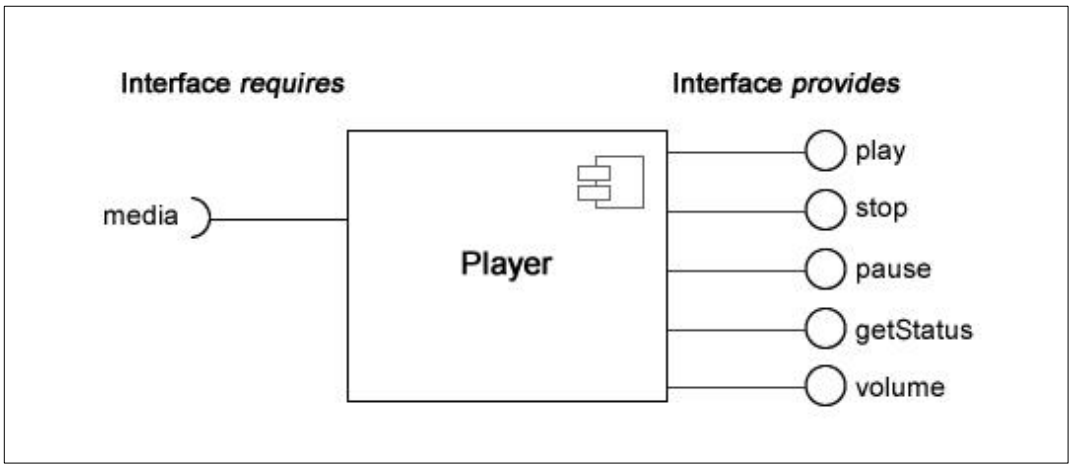


**Figura 4: Exemplo de componente.**

A Figura 4 apresenta o diagrama do componente *Player de mídia*, que é responsável pelo controle de reprodução de uma mídia (como um arquivo de áudio ou vídeo). Este componente necessita de uma entrada (*require*) "*media"*, e fornece interfaces *provides*, que nesse caso são os métodos "*play"*, "*stop"*,

“*pause”* e “*volume”*, para controle da reprodução da mídia. Também fornece o método “*getStatus”*, por meio do qual pode-se saber se a mídia está ou não em execução.

Por mais que se possa observar um componente no sentido dos métodos fornecidos por ele, componentes não podem ser confundidos com objetos pois, diferentemente de classes, que definem objetos, um componente pode estar disponível somente na forma binária e o nome do componente não pode ser utilizado para definir o tipo de uma variável ou parâmetro (WEINREICH e SAMETINGER, 2001, p. 36). Porém, nos dias atuais, a visão de componentes mudou um pouco devido aos novos cenários, existindo, também, componentes em formato caixa branca. Por exemplo, componentes Javascript, CSS ou PHP. Os componentes, tipicamente, interagem entre si por meio de chamadas de métodos, em um estilo cliente/servidor ou publicador/ouvinte (WEINREICH & SAMETINGER, 2001, p. 43).

Apesar de muitos componentes fornecerem métodos, isso não os faz reutilizáveis. Para Sommerviller (2007, p. 293 e 294) os componentes, dentro da Engenharia de *Software*, devem conter algumas características que os fazem reutilizáveis:

- **padronizado:** os componentes devem seguir um modelo, que já foi definido, e devem atender a interface, metadados, documentação, composição e implementação desse modelo. A interface são os *requires* e *provides*. Os metadados são os dados que descrevem o componente, como título e versão. A documentação são os documentos, diagramas e textos que descrevem as funcionalidades do componente, assim como suas dependências. A composição é a forma que os componentes se integram. Já a implementação é a codificação do componente;
- **independente:** o componente não deve necessitar de outros componentes para seu funcionamento, mas caso necessite, isso deve ser especificado na interface *requires*;
- **passível de composição:** para que um componente possa ser composto, suas ações externas devem ocorrer por meio da interface, além disso, suas

informações como métodos e atributos, devem estar acessíveis externamente, para que outros componentes possam acessá-las;

- **implantável:** para um componente ser implantável ele deve ser autocontido e operar sobre o *middleware* que está implementando o modelo;
- **documentado:** os componentes devem ser documentados, dessa maneira, lendo a documentação, os usuários podem decidir se determinado componente serve ou não para resolver determinado problema.

Para se entender estas características, é utilizado o seguinte exemplo: ao se comprar um eletrodoméstico e plugá-lo na tomada, o aparelho irá encaixar e funcionar perfeitamente. Isso porque o plugue do eletrodoméstico e a tomada de energia seguem padrões estabelecidos para que esta conexão seja feita com sucesso. Da mesma forma que existem vários modelos de tomadas ou voltagem, existem também vários modelos de componentes de *software*. Sommerville (2007, p. 295) define modelo de componente como

> [...] padrões para implementação, documentação e implantação de componentes. Esses padrões são para que os desenvolvedores se assegurem de que os componentes podem operar entre si. Eles se destinam também aos fornecedores de infra-estruturas de execução de componentes que fornecem *middleware* para apoiar a operação de componentes (grifo nosso).

Um modelo de componente faz com que as implementações que o seguirem se integrem por meio das interfaces *requires* e *provides*, e se comuniquem. Além disso, define o modelo da documentação que será produzida. Para D'Souza e Wills (1998, p. 390) modelos para componentes são mais abrangentes ainda, definindo, entre outros fatores, empacotamento, ciclo de vida, conectores, interfaces providas e requeridas etc.

Existem alguns elementos, abordados por Weinreinch e Sametinger (2001, p. 38), que são fundamentais para um modelo de componente ideal:

- **interfaces:** especificação do comportamento de componentes e definição de propriedades de linguagens de descrição de interface. Por exemplo, a

IDL (*Interface Description Language*) do CORBA ou WSDL (*Web Services Description Language*) do WebService;

- **nomes:** nomes exclusivos para interfaces e componentes. Em WebServices, por exemplo, a URL é o nome exclusívo do componente;
- **metadados:** informações sobre os componentes, interfaces e seus relacionamentos. No WebService, por exemplo, a WSDL também é utilizada para descreve os serviços/componentes;
- **interoperabilidade:** comunicação e troca de dados entre componentes de diferentes fornecedores, implementados em diferentes linguagens e plataformas. Por exemplo um serviço de cálculo de frete, fornecido pelos correios, desenvolvido em Java e o cliente que o consulta pode ser PHP;
- **customização:** interfaces para personalização dos componentes. Como normalmente componentes são desenvolvidos em caixa preta, as maneiras mais comuns de customizar um componente é por meio das interfaces;
- **composição:** interfaces e regras para a combinação de componentes para criar estruturas maiores;
- **apoio à evolução:** regras e serviços para a substituição de componentes ou interfaces por versões mais novas. No caso do CORBA e WebService, se a nova versão do serviço/componente não compromete a interface, a atualização é automática;
- **empacotamento e desenvolvimento:** implementar e empacotar os recursos necessários para instalação e configuração um componente.

Os modelos de componente não são somente padrões, eles são também base para o *middleware* do sistema que fornece apoio aos componentes. Uma implementação de modelo de componente fornece serviços para auxiliar os componentes, como apresenta a Figura 5.

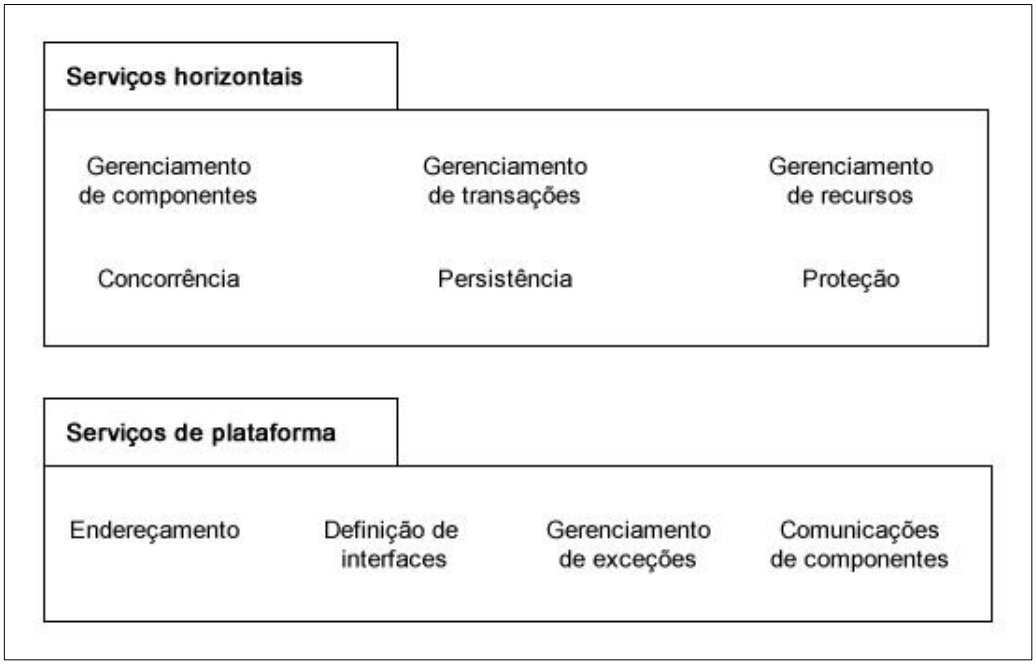


**Figura 5: Serviços fornecidos por um modelo de componente (SOMMERVILLE, 2007, p. 296).**

Na Figura 5 os “serviços de plataforma” são responsáveis pela comunicação entre os componentes:

- **endereçamento:** faz o controle do endereço, ou caminho, em que o componente está disponível. O endereço pode ser local, via rede privada ou *internet*;
- **definição de interfaces:** fornece interfaces *provides* e *requires*, por meio das quais os componentes poderão de comunicar;
- **gerenciamento de exceções:** faz o tratamento de exceções para que o mal funcionamento de um componente não venha a comprometer toda a aplicação;
- **comunicação de componentes:** define e gerencia a comunicação entre os componentes.

Já os “serviços horizontais” são independentes da aplicação e a existência deles pode reduzir os esforços no desenvolvimento e a incompatibilidade entre os componentes:

- **gerenciamento de componentes:** define serviços para criar, excluir, copiar e mover componentes;

- **gerenciamento de transações:** define a coordenação entre componentes assegurando a atomicidade, consistência e o isolamento;
- **gerenciamento de recursos:** ;
- **concorrência:** faz o gerenciamento de transações e *threads* para garantir a consistência e a concorrência dos componentes;
- **persistência:** define uma interface única para o armazenamento persistente do estado dos componentes;
- **proteção:** define serviços de segurança para autenticação, autorização e não-repúdio.

Com a disponibilização desses serviços pelos *middlewares*, torna-se mais fácil a criação e reutilização de componentes. Porém, apesar dessas vantagens, a reutilização de componentes não é imediata. Sommerville (2007, p. 297) afirma que a maioria dos componentes reusados são desenvolvidos dentro da própria empresa e geralmente esses componentes não são imediatamente reusáveis, pois contêm características específicas e, dependendo da necessidade, eles são adaptados. O autor ainda enfatiza algumas ações que devem ser tomadas no processo de adaptação do componente:

- remover os métodos específicos da aplicação;
- mudar os nomes para torná-lo mais genérico;
- adicionar métodos para que se possa obter uma abrangência maior;
- tratar exceções consistentemente em todos os métodos;
- adicionar interfaces de configuração para deixá-lo mais flexível;
- integrá-lo com outros componentes necessários para aumentar a independência.

As empresas que optam pela reutilização de componentes devem, também, adequar seu processo de desenvolvimento de *softwares*. A seção 2.3.2 irá apresentar como é realizada a adequação dos processos de desenvolvimento de *software* tradicionais à CBSE.

### *2.3.2 Processo de CBSE*

Para Sommerville (2007, p. 298) o desenvolvimento de componentes bem sucedido requer um processo de desenvolvimento voltado para o CBSE. Dessa forma, o autor aponta algumas diferenças entre um processo de desenvolvimento de *software* que usa CBSE e um processo de desenvolvimento de *software* que não utiliza CBSE:

1. a análise de requisitos, inicialmente, não é rica em detalhes; os *stakeholders* (pessoas envolvidas no *software*) são encorajados a serem flexíveis, pois funcionalidades muito específicas podem dificultar o reuso;
2. os requisitos são refinados e modificados no início do processo. Caso não se encontre componentes que atendam as necessidades do cliente, outros componentes semelhantes devem ser apresentados, pois a reutilização pode diminui os custos e o cliente pode até mudar de ideia;
3. inicialmente são realizadas buscas de componentes e, após definir a arquitetura do projeto, é realizado um refinamento dos componentes, pois alguns deles podem não funcionar totalmente, ou parcialmente, com a arquitetura proposta;
4. os componentes são integrados com infraestrutura, de acordo com o modelo de componentes, mas algumas funcionalidades precisam ser adicionadas aos componentes, porém esse processo deve ser feito de tal forma que os componentes possam ser reutilizados posteriormente.

A adaptação de um processo de desenvolvimento de *software* existente para o desenvolvimento baseado em componentes pode não ser uma tarefa fácil. É necessário realizar treinamentos para a equipe, adaptação das tecnologias, definição de novas necessidades do processo e documentação do novo processo. Essa documentação se faz necessária para realização de consultas, por membros existentes, ou treinamento de novos membros.

Uma alternativa à adaptação de um processo de desenvolvimento que não é baseado em componentes seria a utilização de processos existentes que já visam essa prática. O *Catalysis*, criado por D'Souza e Wills (1998), fornece algumas técnicas e métodos para o reuso de *software*. Sua abordagem contém,

além do desenvolvimento baseado em componentes, uma visão para projetos orientados a objetos, projetos de alta integridade e reengenharia. De acordo com os autores, o *Catalysis* baseia-se em um número muito pequeno de conceitos UML e segue alguns padrões definido pelo OMG (*Object Modeling Group*), criadores do CORBA.

De acordo com Pressman (2006, p. 243-245) existem quatro princípios básicos no processo de desenvolvimento *softwares* baseado em componentes:

- **Princípio Aberto-Fechado:** "um componente deveria ser aberto para extensão, mas fechado para modificação". Ou seja, um componente deve meios que o desenvolvedor que o utilizar, possa estendê-lo, sem a necessidade de modificar seu código fonte. Isso poderia ser feito, por exemplo, por meio de classes abstratas;
- **Princípio da Substituição de Liskov:** "subclasses devem ser substituíveis por suas classes-bases". Um componente que utiliza uma classe-base deve funcionar com a utilização de uma classe subclasse que estende a classe base. Por exemplo, um componente que requer a classe “Pessoa” deve funcionar se receber a classe “Professor”, que estende pessoa;
- **Princípio da Inversão de Dependência:** "confie nas abstrações. Não confie nas concretizações". Componentes que utilizam classes abstratas têm mais facilidades para serem estendidos sem complicações;
- **Princípio da Segregação de Interface:** "muitas interfaces específicas de clientes são melhores do que uma interface de propósito geral". Em outras palavras, esse princípio aborda a ideia de não criar uma interface genérica que forneça muitas operações. Deve-se criar interfaces específicas para cada categoria de clientes.

Assim como já existe processos de desenvolvimento de software baseado em componentes, também existem modelos de componentes, como o CORBA e o Web Service, que são apresentados na seção 2.4.

## 2.4 Trabalhos Relacionados

Este trabalho tem como objetivo um modelo de especificação de componentes para a *web*. Porém, já existem alguns trabalhos que têm como finalidade a reutilização de componentes, como o CORBA (*Common Object Request Broker Architecture*), que, segundo Pressman (2006, p. 670) é um modelo que fornece diversos serviços para a conexão e comunicação entre componentes independe de sua localização no sistema. De acordo com (RICCIONI, 2000, p. 57) o CORBA permite a execução remota de métodos de objetos.

A comunicação entre componentes abordada pelo CORBA é realizada na rede, por meio uma espécie de “barramento” comum, chamado do ORB (*Object Request Broker*) (GOMES, 2001, p. 2). A Figura 6apresenta o modelo de referência do ORB:

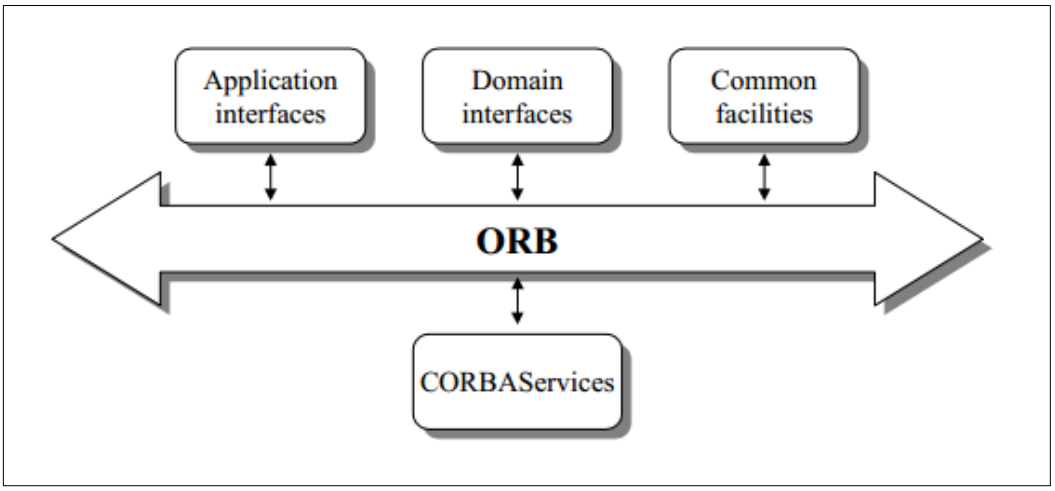


**Figura 6: Modelo de Referência do ORB (GOMES, 2001, p. 2).**

A Figura 6 é apresenta a seguinte estrutura:

- ***Application interfaces*:** são os objetos que podem ser considerados visíveis ao nível de aplicação;
- ***Commom facilities*:** definem facilidades e interfaces no nível de aplicação, como por exemplo, manipulação de dados e armazenamento;
- ***Domain interfaces***: são semelhantes aos recursos horizontais, mas são orientadas para domínios de aplicação específicos;

- **ORB:** *Object Request Broker* é o meio de comunicação entre cliente e servidor;
- **CORBAServices:** definem serviços que ajudam a gerenciar e a manter objetos.

O ORB faz parte de uma arquitetura maior, o CORBA. De acordo com Gomes (2001, p. 1-2), a arquitetura do CORBA define um *framework* para o desenvolvimento de aplicações distribuídas orientadas a objetos. Essa arquitetura permite a invocação de serviços, pelas aplicações clientes, por meio do *middleware*. A Figura 7 apresenta a arquitetura do CORBA.

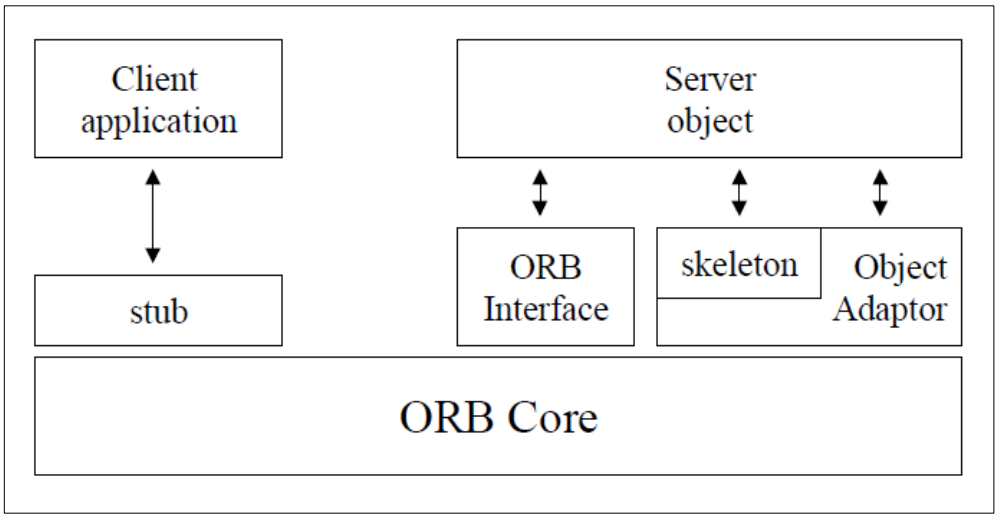


**Figura 7: Arquitetura CORBA (GOMES, 2001, p. 2).**

A Figura 7 apresenta a seguinte estrutura:

- ***client application*:** aplicação que utiliza o modelo do CORBA;
- ***stub*:** faz o empacotamento da chamada do método e desempacotamento do respostas;
- **ORB *interface*:** interface de comunicação definida pelo modelo de referência, como apresentado na Figura 6;
- ***object adaptor*:** fornece um ambiente de execução uniforme para os objetos servidores (*server object*);
- ***skeleton*:** faz o desempacotamento da chamada do método e empacotamento de resposta. É o processo inverso ao *stub*;
- ***server object*:** objeto que contém os métodos a serem invocados;

O cliente obtém a referência de objeto do servidor. O cliente cria uma instância do objeto e invoca um método fornecido pela instância. Essa chamada de método é empacotada pelo *stub* e envia, por meio da rede, pelo ORB *Core* ao servidor. O *stub* é criado a partir do modelo de referência. No servidor, o *skeleton* faz o desempacotamento da invocação do método, passa para o adaptador de objeto que faz a invocação do objeto do servidor. O retorno do método é empacotado pelo *skeleton* e enviado ao cliente. No cliente, o *stub* faz o desempacotamento do retorno e passa para a aplicação cliente.

No ambiente *web*, sobre o protocolo HTTP, esta comunicação entre aplicações é feita utilizando WebServices que, segundo Sommerville (2007, p. 185), disponibilizam informações de uma empresa como forma de serviço, na qual a empresa fornecedora do serviço escolhe quais informações estarão disponíveis e como podem ser acessadas. Para Snell, Tidwell e Kulchenko (2002, p. 1) o conceito de WebService é mais simples do que se imagina, é uma interface de rede acessível à funcionalidade da aplicação, construído utilizando tecnologias padrão da Internet. Em outras palavras, se uma aplicação pode ser acessada através de uma rede utilizando uma combinação de protocolos como HTTP, XML ou SMTP, então é um serviço *web*. A Figura 8 apresenta um exemplo de utilização de WebService.

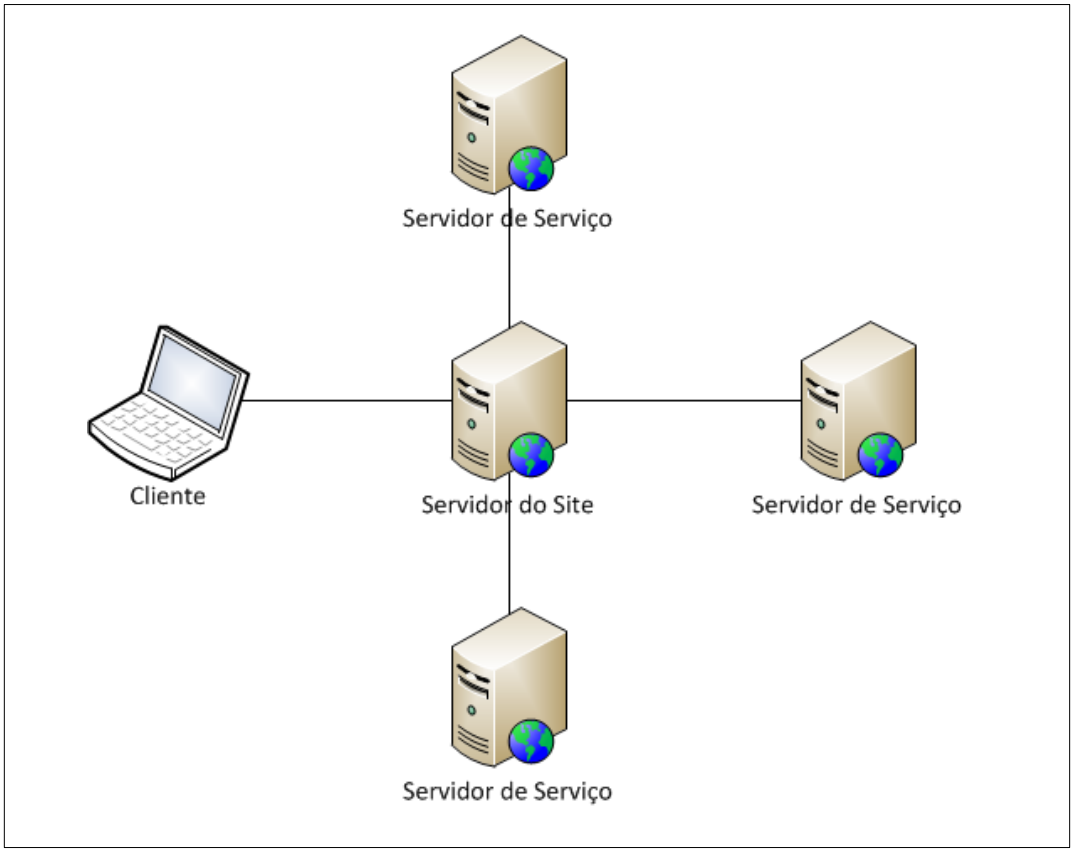


**Figura 8: Exemplo de WebService.**

A Figura 8 apresenta um exemplo de WebService, no qual o cliente faz uma requisição HTTP, por meio do navegador, para o servidor do site, que por sua vez realiza requisições HTTP aos servidores fornecedores de serviços. As requisições dos serviços serão retornadas ao servidor do site, que irá retorna o conteúdo a ser apresentado para o navegador do cliente.

Nas duas formas apresentadas, tanto o CORBA como o WebServices, as aplicações funcionam de modo isolado e a comunicação entre elas é realizada por meio de redes de computadores.

Este capítulo apresentou os conceitos estudados para o desenvolvimento deste trabalho. Com os materiais analisados, pode-se obter o conhecimento necessário sobre o Reuso de *Software*, Engenharia de Domínio e Engenharia de *Software* Baseada em Componentes. Os capítulos seguintes apresentam os materiais e a metodologia utilizada neste trabalho, os resultados obtidos e as conclusões.

# 3 MATERIAIS E MÉTODOS

## 3.1 Materiais

Este trabalho utiliza diversos materiais e ferramentas de software, como livros, dissertações de mestrado, artigos e documentações técnicas de software. Dentre as ferramentas de software estão bibliotecas, *frameworks* e a linguagem de programação PHP. A seguir, cada seção apresenta alguns detalhes da utilização destas ferramentas.

### *3.1.1 PHP*

O PHP é uma linguagem de programação, interpretada, voltada para o ambiente *web*. É uma linguagem de fácil aprendizado, que contém uma sintaxe parecida com a de outras linguagens, como o C, Java e Perl (Manual do PHP, *online*).

O PHP foi criado em 1995 por Rasmus Lerdof, que deu o nome de "*Personal Home Page Tools"*, um conjunto de scripts para guardar estatísticas de acessos de seu site pessoal (Manuel do PHP, *online*). Lerdof compartilhou o seu código-fonte e, posteriormente, em 1997, o PHP foi reescrito por Zeev e Andi e ganhou um novo significado, *"PHP: Hypertext Preprocessor"*, sendo "PHP" um acrônimo recursivo (BACON, 2007, p. 7).

Neste trabalho o PHP é utilizado no desenvolvimento do repositório de componentes e do *middleware*. O repositório é desenvolvido com o Trilado *Framework*, que necessita do PHP.

### *3.1.2 Twitter Bootstrap*

O *Twitter Bootstrap* é um *kit* de ferramentas (ou *toolkit*) *front-end*, criado pelos engenheiros do *Twitter*, para o desenvolvimento rápido para a *web* (OTTO, *online*). O *Twitter Bootstrap* contém uma coleção de padrões de CSS e HTML para o desenvolvimento da interface de formulários, tabelas, botões, entre outros componentes de interface (OTTO, *online*).

Neste trabalho o Twitter Bootstrap é utilizado para o desenvolvimento da interface com o cliente do repositório de componentes.

### *3.1.3 JSON*

O JSON (acrônimo de JavaScript *Object Notation*) é formato de intercambio de dados em texto plano, que apesar de ser baseado em Javascript, foi criado para ser utilizado independente de linguagem (*Introducing* JSON, *online*). Essa notação consegue descrever diferentes tipos de estruturas de dados (como objetos e vetores), seguindo a sintaxe da própria Javascript.

Uma das principais características do JSON é que ele é de fácil compreensão tanto para seres humanos como para máquinas, além disso, seu processamento também é leve (*Introducing* JSON, *online*).

Neste trabalho o JSON é utilizado no desenvolvimento de um componente, na qual o desenvolver utiliza sua sintaxe para criar o arquivo descritor do componente (component.json). O repositório verifica a sintaxe do arquivo descritor está conforme o modelo.

### *3.1.4 Trilado Framework*

O Trilado é um *framework* PHP, que segue o padrão MVC (*Model-View-Controller*) e tem como objetivo o desenvolvido rápido de aplicações para a *web*. Seu nome tem como origem os “três lados” do padrão MVC (NEVES, 2010, p. 128).

Neste trabalho o Trilado *Framework* é utilizado para o desenvolvimento do repositório.

### *3.1.5 RESTful*

REST, acrônimo para *REpresentational State Transfer*, é um conjunto de princípios que define como padrões web, como HTTP e URI, devem ser utilizados (FIELDING, *online*). A arquitetura REST é baseada em quatro princípios (FIELDING, *online*):

1. **identificação dos recursos:** os recursos sempre têm um identificador único: a URI (*Universal Resource Identifier*). Por exemplo, no contexto de um site de notícias, http://site.com/noticia/1 é a URI para a notícia com o identificador “1”;

2. **métodos padrões:** os recursos são manipulados utilizando os verbos padrões protocolo HTTP: GET, POST, PUT e DELETE. Por exemplo, para buscar informações, utiliza-se o GET. Para apagar informações o método DELETE;
3. **recursos independentes de representações:** os recursos são independentes de uma representação, como o HTML, sendo o cliente responsável por informar o formato que ele irá lidar. Exemplo, XML, HTML ou TEXT, na qual o cliente pode informar, ao fazer uma requisição ao serviço, o formato que desejar o retorno, informado o atributo "*Accept*";
4. **comunicação sem estado:** as interações ocorrem por meio de *links* e o servidor não guarda o estado dessas interações. Por exemplo, ao clicar em um *link*, o cliente envia as informações ao servidor. Ao clicar em outro *link*, o cliente manda as informações novamente. Esta característica permite que o servidor atenda um número maior de clientes.

De acordo com Xavier (2011, p. 54) serviços *web* que utilizam REST são implementados de duas formas: Hi-REST e Lo-REST. A Hi-REST é a forma descrita por Fielding (*online*), apresentada anteriormente, que também é conhecida como RESTful. Já a Lo-REST utiliza apenas os métodos GET e POST, pois a maioria dos servidores de *proxy* e *firewall* não permitem a execução dos outros métodos.

O REST é utilizado neste trabalho, como Lo-REST, na API do repositório de componentes. A API é utilizada pelo *middleware* para checagem das versões dos componentes e para *download* dos mesmos.

## 3.2 Metodologia

Esse trabalho foi feito em oito etapas:

1. estudo sobre o tema;
2. elaboração do projeto do TCC;
3. elaboração do modelo de componentes;
4. escolha das tecnologias e ferramentas;
5. desenvolvimento do repositório de componentes;
6. desenvolvimento do *middleware*;

7. definição dos casos de teste;
8. construção de um componente de teste; e
9. escrita dos resultados.

A Figura 9 ilustra visualmente esta metodologia.

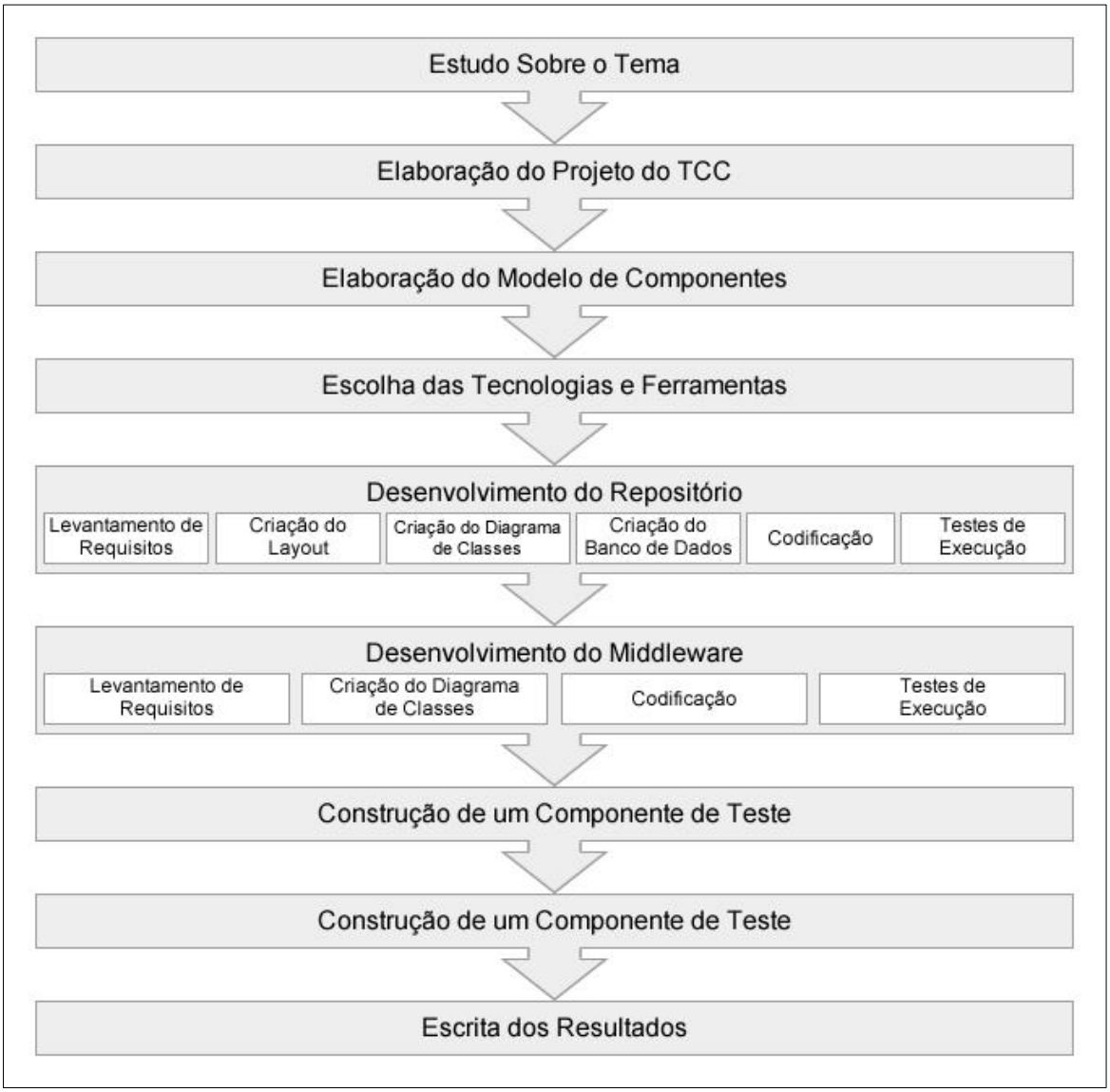


**Figura 9: Metodologia.**

Como mostra na Figura 9, a primeira etapa para o desenvolvimento deste trabalho é caracterizada pela realização do estudo do referencial teórico sobre os assuntos propostos, que envolve Reuso de *Software*, Engenharia de Domínio e Engenharia de *Software* Baseado em Componentes. Sendo assim, foi realizada a busca por livros, monografias, dissertações e teses, em meios físicos ou por meio da internet, da área em questão.

A segunda etapa é constituída da elaboração do projeto do TCC com base no estudo realizado na primeira etapa, onde são representados os objetivos gerais e específicos, justificativa, problema, hipóteses, revisão de literatura e cronograma. Na revisão de literatura são apresentados os conceitos sobre Reuso de Software, apontando suas vantagens e desvantagens. São apresentados, também, os conceitos de Engenharia de Software Baseado em Componentes, que foi a base deste trabalho. Posteriormente são apresentados dois modelos de reuso de componentes já existentes.

Já a terceira etapa é composta pela elaboração do modelo de componentes, tendo como base nos textos desenvolvidos na etapa anterior. Nessa fase são definidas regras que compõem o modelo, como a utilização de um arquivo descritor e quais os atributos esse arquivo descritor deve conter.

Na quarta etapa são escolhidas as ferramentas e tecnologias para a implementação do trabalho. Nesse caso foi definido o PHP e o Trilado *Framework* para o desenvolvimento do repositório e *middleware*. Para a criação da interface optou-se pela utilização do *Twitter Bootstrap*. Para a comunicação entre o *middleware* e o repositório utilizou-se o protocolo REST com JSON. A escolha dessas ferramentas deve-se a facilidade de desenvolvimento, visto que o foco deste trabalho está no modelo de componentes.

A quinta etapa é marcada pelo desenvolvimento do repositório, que está subdivido em outras etapas: 1) levantamento dos requisitos; 2) criação do *layout* do site; 3) criação do diagrama de classes; 4) criação do banco de dados; 5) codificação; e 6) testes de execução. A Figura 10 ilustra essa subdivisão.

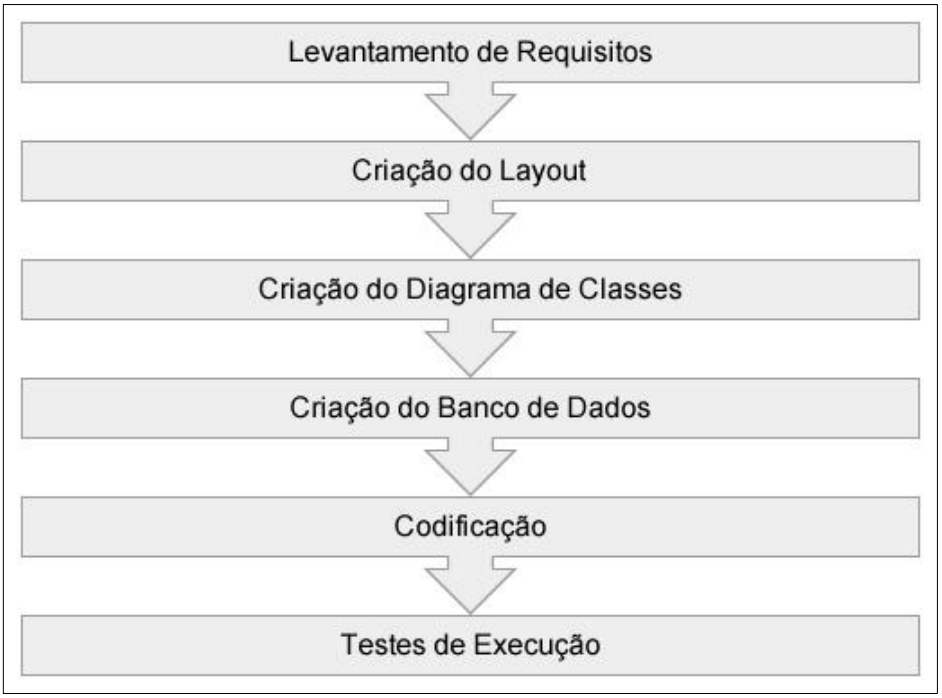


**Figura 10: Metodologia do desenvolvimento do repositório.**

A Figura 10 apresenta a subdivisão da etapa de desenvolvimento do repositório, na qual no levantamento de requisitos é feita a análise das funções que serão disponibilizadas pelo repositório aos usuários. Na criação do *layout* é utilizado o *Twitter Bootstrap* para definir as telas da interface. No diagrama de classes são planejadas as classes que posteriormente são implementadas. Na parte de criação do banco de dados são definidas as entidades que serão utilizadas. A parte de codificação consiste na criação do código com base no diagrama de classes e na interface. Por fim, nos testes de execução, são realizados cadastros, acessos ao sistema (*logon*) e envio de componentes para verificar o funcionamento do repositório. Esses testes são informais, apenas para verificar se há erros de sintaxe que impendem a execução do repositório.

A fase posterior tem foco no desenvolvimento do *middleware*, que é responsável por consultar o repositório, fazer *download* dos componentes e atualizar os arquivos na aplicação que está utilizando o modelo. Essa etapa também está subdivida em: 1) levantamento dos requisitos; 2) criação do diagrama de classes; 3) codificação; e 4) testes de execução. A Figura 11 ilustra a subdivisão:

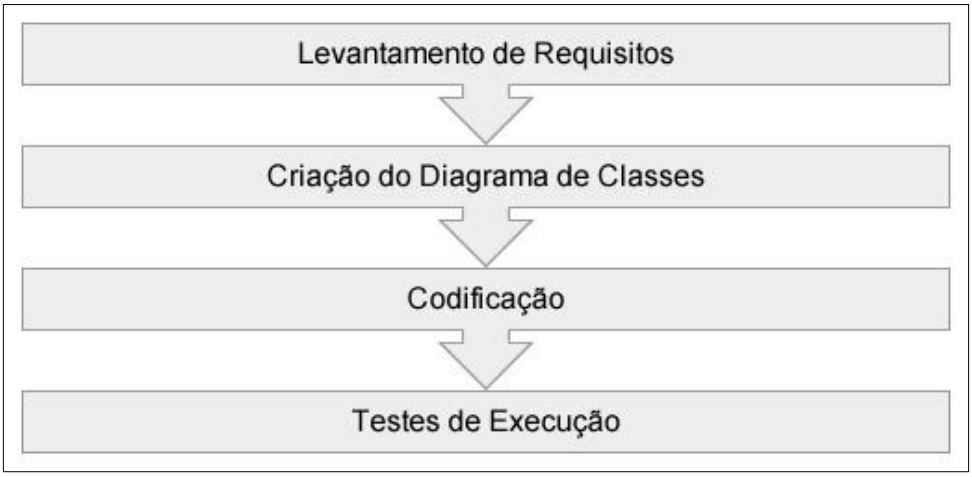


**Figura 11: Metodologia do desenvolvimento do *middleware*.**

A Figura 11 apresenta a subdivisão da etapa do desenvolvimento do *middleware*, que inicia com o levantamento dos requisitos, para poder definir o que o *middleware* oferecerá à aplicação que o utilizará. Na sequência, na parte da criação do diagrama de classes, planejam-se quais classes compõem o *middleware*, assim como seus métodos. Na codificação são implementadas as classes do *middleware*. Por fim, são realizados testes de execução para verificar se contém algum erro no *middleware*. Esses testes são informais, apenas para verificar se há erros de sintaxe que impendem a execução do repositório.

A sétima etapa consiste na definição dos casos de testes. Na qual são elaboradas formas de realização dos testes e quais os requisitos são testados.

Na oitava etapa está a construção de um componente para validar o funcionamento do modelo. Também são realizados testes, na qual os testes consistem nos processos de submissão do componente ao repositório e posteriormente o *download* por meio do *middleware*.

Por fim, realizou-se a escrita sobre o modelo criado, o processo de implementação do repositório e o *middleware*, e a realização dos testes, assim como seus resultados.

Esta metodologia é importante para o êxito do trabalho, pois define as etapas, e a sequência, que são executadas para o desenvolvimento deste trabalho.

# 4 RESULTADOS E DISCUSSÃO

Esta seção apresenta os resultados obtidos como desenvolvimento do trabalho. Além disso, mostra o modelo desenvolvido para a especificação de componentes para a *web*, a criação de um repositório para armazenamento destes componentes e relaciona os conceitos apresentados no capítulo 2 com resultados práticos.

## 4.1 O Modelo de Componentes

De acordo com os estudos realizados, definiu-se um modelo de componentes que pudesse ser aplicado em diversas plataformas de desenvolvimento *web*. O modelo contém um conjunto de especificações para os desenvolvedores de componentes e um conjunto de especificações para o *middleware*. Possui, também, um repositório para armazenamento desses componentes, no qual os desenvolvedores pudessem enviar, buscar, visualizar informações ou realizar *download* de componentes.

Apesar de que, neste trabalho, utilizaram-se exemplos e códigos em PHP, o modelo, como um todo, pode ser aplicado em outras plataformas, como Ruby ou Python. Para isso será necessário o desenvolvimento de um *middleware* compatível com a plataforma. O repositório, também desenvolvido em PHP neste trabalho, suporta componentes de outras linguagens, mas também pode ser implementado para outras plataforma. O modelo, independente da plataforma, contém a arquitetura apresentada na Figura 12.

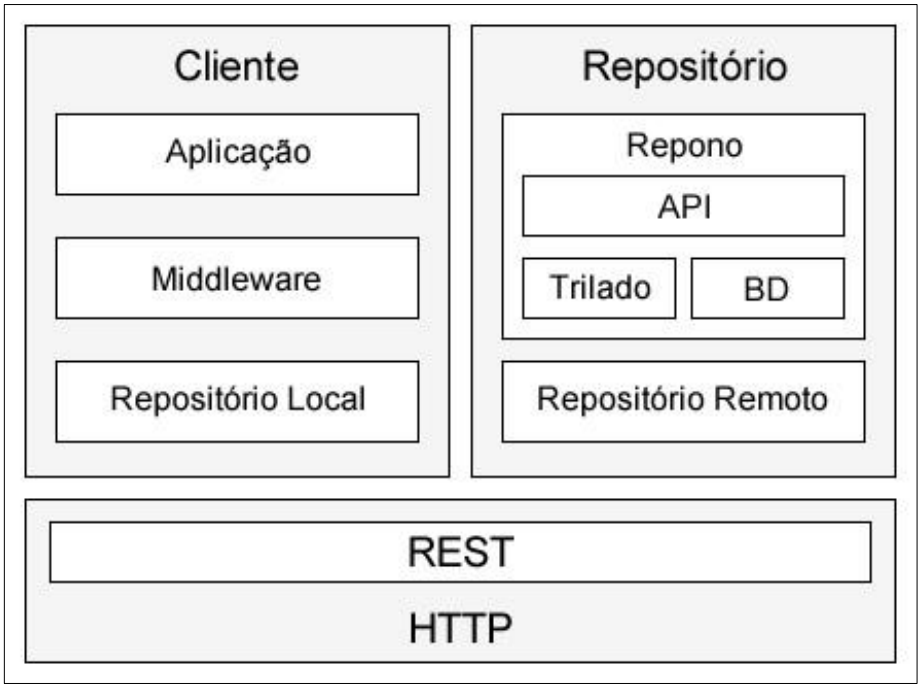


**Figura 12: Arquitetura do Modelo de Componentes criado.**

Conforme ilustra a Figura 12 a arquitetura é composta pelo repositório de componentes e o cliente. O cliente é composto por: repositório de componentes local, *middleware* e *software* do usuário. Já o repositório de componentes é composto por: repositório remoto e pela aplicação de gerenciamento dos componentes (denominado *Repono*). O *software* do repositório utiliza um banco de dados para guardar os metadados dos componentes. Utiliza, também, o Trilado *Framework* para o desenvolvimento da API. A comunicação entre o cliente e o repositório utiliza o protocolo REST, que, por sua vez, funciona sobre o protocolo HTTP.

Após a definição da arquitetura do modelo de componentes, realizou-se um estudo geral de como funcionaria o processo de integração de um componente fornecido pelo repositório e uma aplicação que utiliza os componentes. Dessa maneira, criou-se uma visão geral desse fluxo, para obter um entendimento melhor da comunicação entre o cliente e o repositório. Esse fluxo vai desde a submissão do componente pelo criador até o momento de sua utilização por uma aplicação que segue o modelo. Essa utilização consiste no *download* e na atualização dos arquivos no cliente, por meio do *middleware*, que busca essas informações no servidor. A Figura 13 ilustra este processo de comunicação.

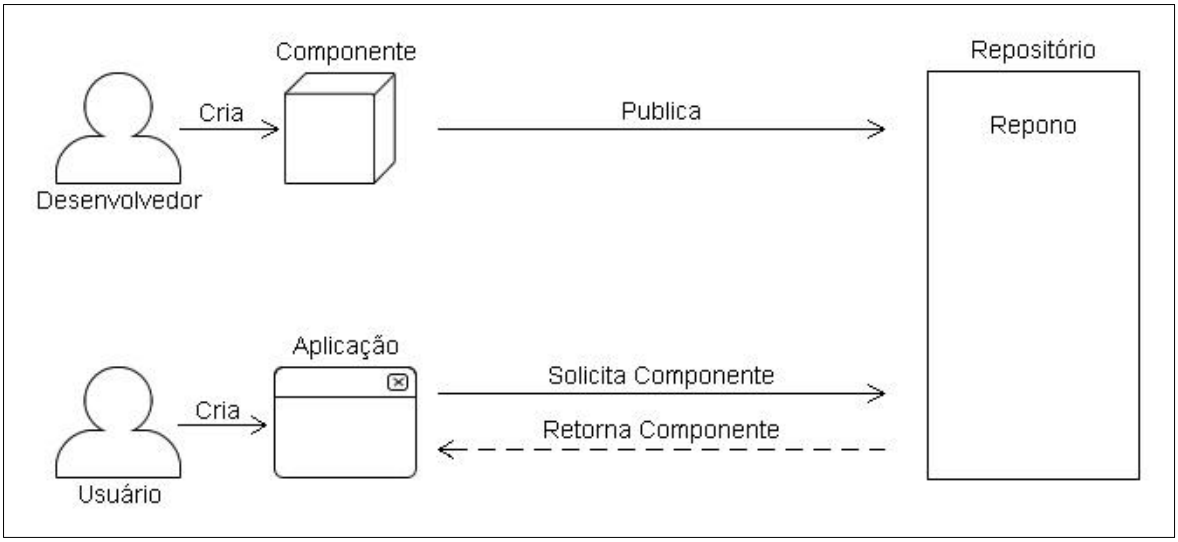


**Figura 13: Visão geral do processo.**

Na Figura 13 o *desenvolvedor* é quem cria um componente e o publica no *repositório*. O *usuário* é quem utiliza o componente para criar uma aplicação. A imagem representa uma sequência de passos que explica o fluxo desde a publicação de um componente até o momento do *download*:

1. O desenvolvedor cria o componente, de acordo com o Modelo de Especificação de Componentes, o compacta e publica no repositório de componentes;
2. O usuário, ao criar uma aplicação, define quais componentes serão utilizados;
3. A aplicação, no momento da execução, executa o *middleware*, que faz requisições ao repositório para saber quais componentes precisam ser baixados ou atualizados;
4. O repositório verifica se há a necessidade de atualização de algum componente, se for necessário, envia-o ao pacote de classes cliente, que faz a atualização dos arquivos localmente.

Portanto, esta arquitetura define quais elementos estão contidos no modelo e a separação entre eles. Os elementos que compõem o cliente, dentro da arquitetura apresentada, são detalhados na seção 4.3.

## 4.2 Modelo de Especificação de Componentes

O modelo de especificação de componentes é um conjunto de regras que define como um componente deve ser criado, documentado, disponibilizado e utilizado.

Para este trabalho, utilizou-se o PHP para criação do repositório, do *middleware* e do componente de teste. Porém, este modelo de especificação de componentes pode ser utilizado com qualquer linguagem, necessitando apenas que o *middleware* esteja de acordo com modelo de especificação definido. O repositório, apesar de ser desenvolvido em PHP, suporta componentes de outras plataformas, desde que sigam este modelo de especificação.

Este modelo de especificação define um conjunto de critérios, os quais devem ser seguidos pelos componentes que seguem o modelo:

- O componente deve estar dentro de um diretório, o qual deve conter o nome do componente, que deve ser único no repositório;
- O componente deve estar acompanhado de um arquivo que o descreve, o qual deve estar dentro do diretório do componente com o nome “component.json”;
- Ao ser enviado para o repositório, o diretório deve ser compactado, usando o formato ZIP.

O arquivo “component.json” segue a ideia apresentada da seção 2: os componentes devem conter *requires*. Este arquivo contém metadados que descrevem o componente e quais as suas dependências. O conteúdo do arquivo está no formato JSON, com propriedades e valores que descrevem o componente. Essas propriedades estão descritas na Tabela 1.

**Tabela 1: Propriedades do arquivo descritor de um componente.**

| **Propriedade** | **Descrição** |
|---|---|
| name | Identificador do componente. |
| title | Título do componente. |
| description | Descrição do componente. |
| version | Versão do componente. |

| authors | Lista dos autores do componente. |
|---|---|
| requires | Lista de outros componentes que este depende. |

Na Tabela 1 são apresentados os metadados que descrevem o componente:

- **name:** identificador único do componente, pode conter apenas letras, números e *underline*. Além disso, deve conter no mínimo 3 e no máximo 16 caracteres. Todas as versões de um mesmo componente devem conter o mesmo identificador;
- **title:** título do componente;
- **description:** texto que resume a utilidade ou a aplicação do componente;
- **version:** versão do componente. O formato aceito é “n.n.n”, sendo “n” qualquer número inteiro positivo;
- **authors:** lista de objetos que contêm os nomes e e-mails dos autores do componente;
- **requires:** lista de objetos que contêm o nome único e a versão de outros componentes que este necessita para o funcionamento. Caso esses componentes não estejam na aplicação, também são baixados e instalados (detalhes desta operação serão vistos posteriormente).

A Listagem 1 mostra um exemplo de um arquivo `component.json` para a criação de um componente, conforme foi definido no modelo de especificação. Nesse caso, o componente criado é o *“Form Helper”*, uma classe, já existente, para auxiliar na geração de formulários do Trilado *Framework*.

```
{
     "name": "trilado_form",
     "title": "Form Helper",
     "description": "Classe para auxiliar na geração de formulário
HTML",
     "version": "1.0.0",
     "authors": [
          {
               "name": "Van Neves",
               "email": "vaneves@vaneves.com"
          }
     ],
     "requires": [
```

```
            {
                  "name": "trilado_html",
                  "version": "1.0.0"
            }
      ]
}
```

**Listagem 1: Exemplo do arquivo descritor.**

Na Listagem 1, o código contém um objeto JSON com as propriedades que foram definidas na Tabela 1. As propriedades `authors` e `requires` são vetores de objetos.

Para o componente ficar completo, de acordo com o modelo de especificação de componentes, ele deve conter, além do arquivo "component.json", o diretório raiz, com o mesmo nome do componente, e estar compactado no formato ZIP. A Figura 14 apresenta a árvore de arquivos e pastas do componente *"Form Helper"*.


**Figura 14: Árvore de arquivos e pastas do componente *Form Helper*.**

Na Figura 14 apresenta a estrutura de arquivos e pastas do componente *Form Helper*. O componente é compactado em um arquivo em formato ZIP, dentro dele há um diretório com o nome único do componente. Dentro do diretório há o arquivo descritor ("component.json") e os arquivos do componente, nesse caso, somente o "Form.php". Apesar de que nesse exemplo o arquivo "Form.php" contém o nome parecido com o nome do componente, não uma regra que defina essa relação. Por exemplo, poderia haver outros arquivos PHP com nomes variados.

Para que um componente seja publicado no repositório ele deve seguir este modelo. No ato da publicação são realizadas verificações, do lado do servidor, a fim de constatar se o componente atende os requisitos exigidos pelo modelo.

## 4.3 Cliente

Nesse contexto o cliente é composto pela aplicação, o middleware e o repositório local. Os elementos deste conjunto ficam no mesmo ambiente, como o computador do desenvolvedor da aplicação ou servidor de produto. As seções 4.3.1, 4.3.2 e 4.3.3 detalham cada item.

### *4.3.1 Repositório Local*

O repositório local é o diretório que armazena os componentes, do lado do cliente, que são baixados pelo *middleware*. Esse repositório existe para que não seja necessário o *middleware* efetuar o *download* dos arquivos de um componente toda vez que a aplicação que o utiliza for executada. O desenvolvedor do *software* define, no arquivo de configuração (é apresentado com detalhes na seção 4.3.2), qual o endereço do repositório local. A Listagem 2 apresenta o código de configuração do endereço do repositório local.

```
Config::set('directory', 'components/');
```

**Listagem 2: Configuração do endereço do repositório local.**

A Listagem 2 apresenta o código de configuração do endereço do repositório local. O endereço informado pelo desenvolvedor do *software* deve estar com permissão de escrita, para que o *middleware* possa alocar os componentes baixados.

A existência desse repositório local é uma das grandes diferenças entre o modelo deste trabalho, o CORBA e o WebService. No caso do CORBA e WebService os componentes são baixados somente arquivos de interface, sendo que a aplicação realiza requisições ao servidor, sempre que for utilizar o serviço/componente, fornecendo entradas. O servidor retorna a saída à aplicação solicitante. O processamento do componente ocorre de forma isolada e a aplicação, nem o desenvolvedor, têm acesso ao código-fonte. No modelo de

especificação de componentes criado neste trabalho os componentes são baixados, por completo, e executados localmente (junto ao *software* do cliente). Só são realizadas novas requisições ao repositório para chegar atualizações dos componentes. O *middleware*, que faz o gerenciamento dos componentes no repositório loca, é apresentado na seção 4.3.2.

### *4.3.2 Middleware*

O *middleware* é um conjunto de classes que deve estar contido no cliente que utiliza o modelo. Essas classes são disponibilizadas para *download* no site do *Repono* e são responsáveis pelo *download* e atualização dos componentes no *Repono*. Elas são responsáveis também pela incorporação dos componentes no repositório local. A Figura 15 mostra o diagrama das classes incluídas no *middleware*.

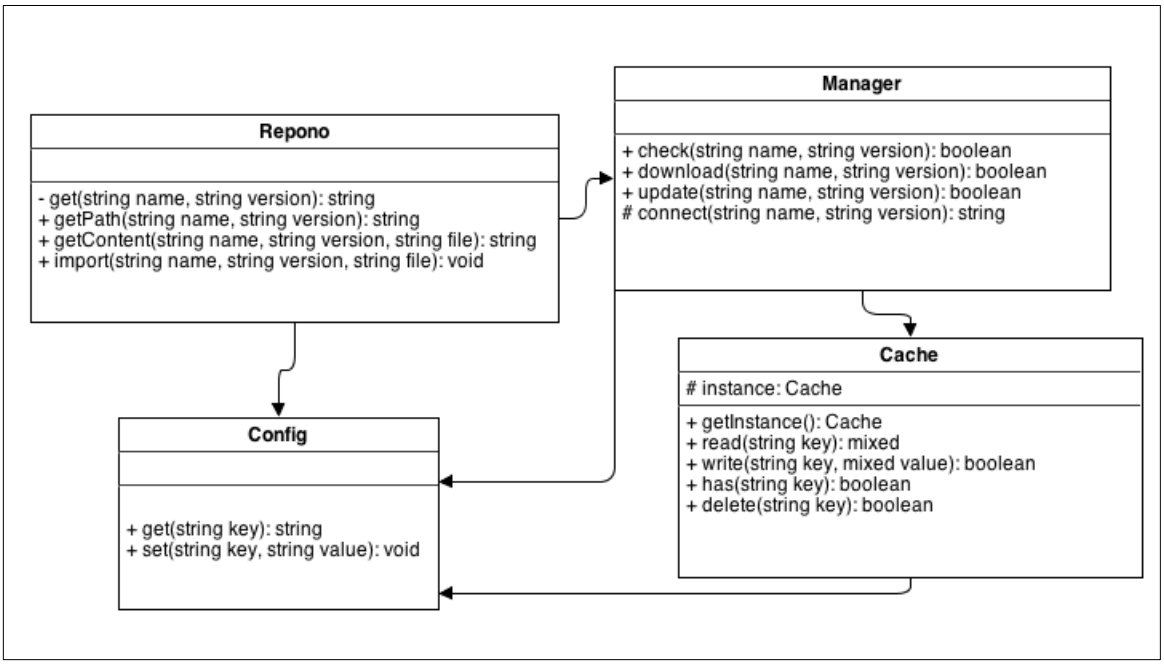


**Figura 15: Diagrama de classes do *middleware*.**

Conforme a Figura 15, as classes são responsáveis pelo gerenciamento dos componentes e pela interação com o repositório. Elas estão definidas da seguinte maneira:

- **`Repono`:** classe que contém os métodos para utilização dos componentes dentro da aplicação:

  - **`get()`:** método estático, privado, utilizado para obter informações sobre um determinado componente;
  - **`getPath()`:** método estático utilizado para obter a URL de um componente dentro do software que o utiliza;
  - **`getContent()`:** método estático utilizado para obter o conteúdo de um arquivo de um componente;
  - **`import()`:** método estático utilizado para importar um arquivo do componente que será executado junto ao *software*;
- **`Manager`:** classe que contém os métodos estáticos para interação com a API do repositório:
  - **`check()`:** verifica se existe uma versão mais recente do servidor;
  - **`download()`:** baixa a versão especificada de um componente no repositório, descompacta os arquivo no diretório especificado e informa se a operação foi realizada com sucesso;
  - **`update()`:** baixa a versão mais recente de um componente no repositório, descompacta os arquivo, substitui pelos existentes no diretório e informa se a operação foi realizada com sucesso;
  - **`connect()`:** faz a conexão com servidor do repositório. É utilizada pelos métodos anteriores;
- **`Cache`:** guarda as informações referentes aos componentes em arquivos de *cache*, diminuindo o fluxo de conexão com o repositório. Essa é a classe de *cache* utilizada por padrão pela classe `Manager`, mas o usuário pode definir outra classe na configuração. Seus métodos foram definidos como:
  - **`getInstance()`:** é o único método estático da classe, trabalha com a propriedade `instance` e retorna uma instância da própria classe. Foi definido seguindo o padrão *Singleton* para garantir uma única instância da classe da aplicação;
  - **`read()`:** retorna os dados que estão no *cache* de uma chave específica;
  - **`write()`:** grava os dados no *cache* em uma chave específica pelo tempo que for determinado;

    - **`has()`:** verifica se existe dados no *cache* em uma chave específica;
    - **`delete()`:** remove os dados do *cache* de uma chave específica;
- **`Config`:** utilizada para realizar a comunicação da configuração entre a aplicação do usuário e as outras classes do *middleware*. Possui apenas dois métodos:
    - **`set()`:** define uma propriedade da configuração. Recebe a propriedade e o valor como parâmetro;
    - **`get()`:** retorna o valor de uma propriedade da configuração. Recebe o nome da propriedade como parâmetro.

Além dessas classes, o pacote cliente do *Repono* constitui-se ainda de um arquivo de configuração, no qual o desenvolvedor, ao utilizar as classes, deve informar o tempo de *cache* das informações, qual o endereço do repositório (pois podem haver outros) e se a atualização dos componentes deve ser feita automaticamente. A Listagem 3 apresenta um exemplo do arquivo de configuração.

```
use \Repono\Config;

Config::set('directory_component', 'components/');
Config::set('directory_cache', 'components/_cache/');
Config::set('time_cache', 30);
Config::set('auto_update', true);
Config::set('url_repository', 'http://repositorio.com /');
Config::set('url_component', 'http://meusite.com/components/');
```

**Listagem 3: Exemplo do arquivo de configuração.**

A Listagem 3 apresenta um exemplo do arquivo de configuração do *middleware*. O desenvolvedor do *software*, que utilizar o *middleware*, pode criar um arquivo para definição das configurações do *middleware*. Em outras palavras pode-se dizer que ele tem a liberdade de escolher em qual arquivo deseja definir as configurações. Por exemplo, o desenvolvedor pode importar a classe “\Repono\Config” dentro do arquivo “settings.php”, “init.php”, “run.php” ou qualquer outro de sua escolha, basta definir as configurações como mostra a Listagem 3. As propriedades aceitas na configuração são apresentadas na Tabela 2.

**Tabela 2: Propriedade de configuração.**

| Propriedade | Descrição |
|---|---|
| **directory_component** | Diretório, no disco, que os componentes serão guardados. |
| **directory_cache** | Diretório, no disco, que o cache será gravado. |
| **time_cache** | Tempo, em minutos, de cache. |
| **auto_update** | Valor “booleano” (*true* ou *false*) que define se os componentes serão atualizados automaticamente quando houver novas versões. |
| **url_repository** | URL do repositório remoto. |
| **url_component** | URL do repositório local. |

A Tabela 2 apresenta as propriedades que são definidas no arquivo de configuração.

Este trabalho apresenta um *middleware* desenvolvido em PHP, portanto, só poderá ser utilizado por aplicações também em PHP. Porém, seguindo as especificações apresentadas nas seções 4.2, 4.3.1, 4.3.2 e 4.4.4 é possível programar um *middleware* para outras plataformas.

### *4.3.3 Aplicação*

A aplicação é o *software* que utiliza componentes baseados no modelo de especificação de componentes. A linguagem ou a plataforma utilizadas não são limitadas, a não ser à compatibilidade com o *middleware*.

A aplicação é responsável por importar o *middleware* e executar as chamadas de componente. A Listagem 4 apresenta um exemplo de como importar o *middleware* na aplicação.

```
require_once './repono/Config.php';
require_once './repono/Cache.php';
require_once './repono/Manager.php';
require_once './repono/Repono.php';
```

**Listagem 4: Exemplo de importação do *middleware*.**

A Listagem 4 apresenta um exemplo de importação do *middleware*. No caso, a função do PHP `require_once` inclui os arquivos do *middleware*, que estão no diretório “repono”. O diretório também pode ser definido pelo desenvolvedor do *software*.

Na aplicação devem ser realizadas, também, as importações de componentes, sejam eles de interface ou não. A importação é realizada utilizando as classes do *middleware*, na qual, oferecem tais métodos, estáticos, para a importação. A Listagem 5 apresenta um exemplo de importação de componente.

```
<script src="<?= Repono::getPath('jquery') ?>/jquery.js"></script>
```

**Listagem 5: Exemplo de importação de componente Javascript.**

A Listagem 5 contém o código exemplo de importação de um componente em Javascript. Nesse caso o Javascript se trata da biblioteca jQuery[4], que já existe, porém transformado em componente de acordo com modelo de especificação. O método `Repono::getPath()`, utilizando na Listagem 5, retorna a URL do componente, no caso o jQuery, dentro da aplicação. Essa URL pode ser utilizada para importar o Javascript de acordo com sua *tag* HTML.

O método `Repono::getPath()`, também pode ser utilizado para importar arquivo CSS, como mostra a Listagem 6.

```
<link rel="stylesheet" href="<?= Repono::getPath('example')
?>/file.css">
```

**Listagem 6: Exemplo de importação de componente CSS.**

Na Listagem 6 é importado o arquivo “file.css” do componente “example”. Além desse método, a classe “Repono”, do *middleware*, fornece mais duas forma de importação de componentes: componentes HTML e arquivo de execução PHP.

Os componentes HTML, que pode conter trechos da interface do usuário, podem ser importados utilizando o método `Repono::getContent()`, como mostra a Listagem 7.

[4] Biblioteca Javascript para manipulação de HTML.

```
<?= Repono::getContent('example', '1.0.0', 'file.html') ?>
```

**Listagem 7: Exemplo de importação de componente HTML.**

A Listagem 7 apresenta um exemplo de importação de um componente HTML. Nesse caso são informados o nome do componente, a versão e o nome do arquivo que deseja utilizar. A passagem do nome do arquivo como parâmetro foi definido por causa dos componentes que contêm mais de arquivo. Portanto, é obrigatório informa-lo. Esse método retorna o conteúdo do arquivo para ser impresso na tela.

Quando se trada de um arquivo PHP necessitará ser executado, pois nesse caso, o interessante é o retorno da execução do arquivo, não seu conteúdo. Para isso, deve-se utilizar o método `Repono::import()`, que recebe, além do nome e versão do componente, o nome do arquivo a ser executado. A Listagem 8 apresenta um exemplo de importação de componente PHP.

```
<?php Repono::import('example', '1.0.0', 'file.php') ?>
```

**Listagem 8: Exemplo de importação de componente PHP.**

A Listagem 8 apresenta o exemplo de importação de um componente com arquivo que necessita ser executado. Nesse caso o método só funcionará para arquivos PHP porque o *middleware* é em PHP. No caso *middleware* desenvolvido para outra plataforma, os arquivos devem ser de acordo com as regras da plataforma.

## 4.4 Repositório de Componentes

Para que a reutilização dos componentes desenvolvidos baseados no modelo proposto neste trabalho se torne realidade, é necessário que esses componentes possam estar amplamente disponíveis e acessíveis a qualquer momento. Portanto, a criação de um repositório de componentes possibilita a centralização, o armazenamento e o controle de cada uma das versões desses componentes.

Neste trabalho desenvolveu-se um repositório para armazenamento de componentes de acordo com o modelo proposto. A seção 4.4.1 descreve o repositório.

### *4.4.1 Repono*

O *Repono* (pronuncia-se “répono”) é um repositório desenvolvido para o ambiente *web* que permite o armazenamento, a pesquisa e a recuperação de componentes que seguem um modelo definido neste trabalho. “Repono” é uma palavra do Latim, que significa “armazenamento”.

O *Repono* é um repositório público, por meio do qual qualquer pessoa pode ter acesso aos componentes publicados. Porém, para publicação de um componente é necessário a realização de um cadastro, confirmação e autenticação com usuário e senha. Não há necessidade de autenticação para recuperação de componentes.

As funcionalidades definidas para o repositório são:

- **cadastro de usuário:** permite ao visitante do repositório realizar seu cadastro;
- **confirmação do cadastro:** permite ao visitante confirmar seu cadastro;
- **publicação de componente:** permite ao usuário publicar componentes;
- **atualização de componente:** permite ao usuário que envie uma nova versão de um componente;
- **pesquisa de componentes:** permite que os usuários ou visitantes realizem pesquisas por componentes;
- **visualização de componente:** após realização de uma pesquisa, os usuários podem visualizar as informações que descrevem um determinado componente;
- **baixar componente:** tanto o usuário como o visitante podem realizar o *download* de qualquer componente;
- **documentação:** permite ao usuário aprender sobre criação e utilização de componentes de acordo com o modelo que foi definido.

### *4.4.2 Repositório Remoto*

O repositório remoto é o diretório, dentro do servidor do *Repono*, onde são armazenados os componentes. Esses componentes são compactados antes do envio, pelo o desenvolvedor, conforme o modelo de especificação de

componentes, e não são executados no servidor, pois estão disponíveis apenas para *download*.

No repositório remoto podem ficar várias versões de um mesmo componente. No ato da submissão, o *Repono* salva o arquivo compactado com o nome do componente e versão. Por exemplo, o componente jQuery, com a versão 1.9.1, ficaria “jquery-1.9.1.zip”. Dessa maneira, não haverá conflito de nomes de arquivos.  A seção 4.4.3 apresenta a implementação que faz esse gerenciamento dos componentes no *Repono*.

### *4.4.3 Implementação*

Para a implementação do repositório utilizou-se o Trilado *Framework*, que foi apresentado da seção 3.1.4. Criou-se um diagrama de classes para melhor definição e visualização do sistema, o qual é apresentado pela Figura 16.

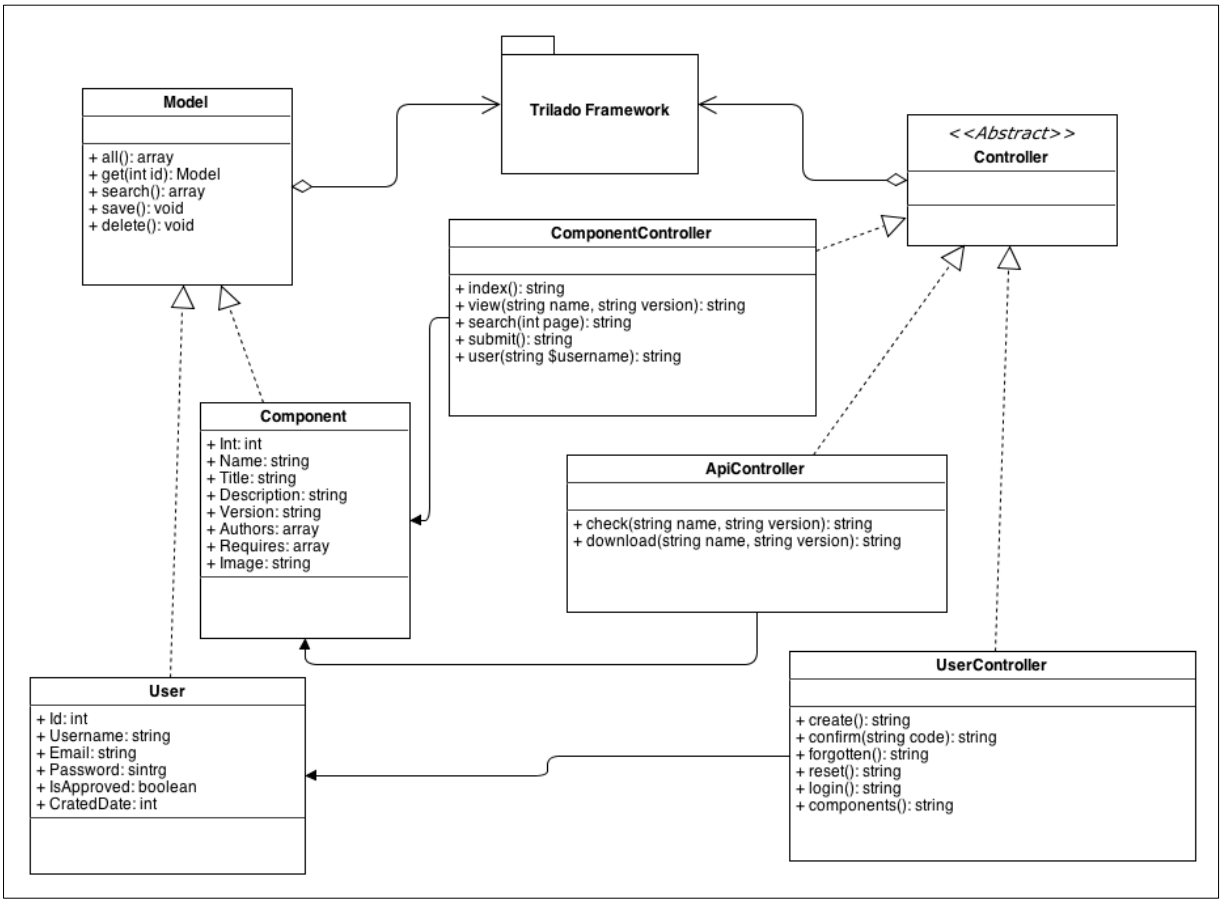


**Figura 16: Diagrama de classes do Repono.**

A Figura 16 contém as classes criadas para a implementação do *Repono*. O pacote “Trilado *Framework*” representa o *framework* propriamente dito. As classes estão definidas da seguinte maneira:

- **`Controller`:** faz parte do pacote do *framework*. É abstrata e deve ser herdada pelos demais *controllers*;
- **`Model`:** faz parte do pacote do *framework*. Representa uma entidade do banco de dados e deve ser herdada. Contém os principais métodos de manipulação de banco de dados, seguindo o padrão *Active Record*;
- **`User`:** herda da classe `Model`. Representa a entidade usuário no banco de dados;
- **`Component`:** herda da classe `Model`. Representa a entidade componente no banco de dados;
- **`UserController`:** herda da classe `Controller`. Contém métodos que representam páginas, como:
  - **`create()`:** página de auto cadastro;
  - **`confirm()`:** página de confirmação do cadastro;
  - **`forgotten()`:** página de recuperação de senha;
  - **`reset()`:** página para redefinição de senha;
  - **`login()`:** página de autenticação;
  - **`components()`:** página de listagem dos componentes de um determinado usuário;
- **`ComponentController`:** herda da classe `Controller`, contém métodos que representam as páginas sobre os componentes, como:
  - **`index()`:** página inicial do repositório;
  - **`search()`:** página de pesquisa de componentes;
  - **`view()`:** página de visualização de um componente;
  - **`submit()`:** página para publicação de um componente;
  - **`user()`:** página par listagem dos componentes pertencentes à um usuário;
- **`ApiController`:** herda da classe `Controller`, fornece métodos para as classes das aplicações clientes consultarem o repositório. Os métodos são:

- **`check()`:** método para verificar se a versão do componente que está no cliente é a mais atual e precisa ser atualizada;
- **`download()`:** método para baixar uma versão específica do componente.

Na criação das *views* (interface gráfica) optou-se pela utilização do *Twitter Bootstrap*, apresentado da seção 3.1.2, devido a seus recursos para a criação de interfaces de páginas *web*. A Figura 17 apresenta a definição da tela de visualização de um componente.


**Figura 17: Página de visualização de componente.**

A Figura 17 apresenta a página de visualização de um componente que está publicado no *Repono*. Algumas informações contidas na página são extraídas do arquivo descritor do componente no ato da sua publicação, como: título, versão, descrição, autores e dependências. Os e-mails dos autores também estão contidos no arquivo descritor, portanto, as fotos dos mesmos são obtidas do

Gravatar[5]. As dependências fazem *link* para a página de visualização dos componentes. O exemplo de utilização mostra um código que pode ser utilizado em uma aplicação que utilizará o componente. O histórico de versões é gerado automaticamente à medida que o usuário envia uma nova versão do componente.

### *4.4.4 API (Application Programming Interface)*

A API do *Repono* fornece serviços, utilizando REST, para que o *middleware* realize consultas ao gerenciador do repositório. A API fornece apenas dois serviços: checagem de versão e *download* o componente.

Como apresenta a seção 4.3.2, é possível definir, no arquivo de configurações, qual diretório o *middleware* irá consultar. Para que o diretório possa fornecer esse serviço, é necessário que ele siga, também, o modelo de especificação da API. Essas especificações consistem basicamente no formato de entrada e saída dos serviços de checagem de versão e *download* do componente:

- **Checagem de versão:** deve estar a partir do endereço do repositório com "/api/check/name/version", sendo "name", o nome único do componente e "version" a versão do componente. Por exemplo, "api/check/jquery/1.9.1". Deve retorna um número inteiro, sendo:
  - **zero:** caso a versão do componente informada seja igual a versão mais recente presente no repositório;
  - **maior que zero:** caso exista uma versão mais recente no repositório;
  - **menor que zero:** caso a versão informada seja mais recente que as versões contidas no repositório;
  - caso o componente não seja encontrado, é retornado o erro HTTP 404;
- **Download de componente:** deve estar a partir do endereço do repositório com "/api/download/name/version", sendo "name", o nome único do componente e "version" a versão do componente. Por exemplo, "/api/check/jquery/1.9.1. Deve retorna o arquivo ZIP do componente, de

[5] **G**lobally **R**ecognized **Avatar**: http://pt.gravatar.com/

acordo com os dados informados. Caso a versão não seja informada, deve retorna a última versão do componente. Caso o componente não seja encontrado, deve retornar o erro HTTP 404.

## 4.5 Validação

Para verificar o funcionamento do modelo de especificação de componentes criado, foram criados componentes seguindo o modelo. Posteriormente, os componentes foram enviados ao *Repono*, para *download* e atualização por meio do *middleware*. Também foram enviados componentes inválidos, para verificar o comportamento do repositório. Além disso, houve a tentativa de utilizar componentes inexistentes no repositório.

O primeiro componente criado consiste na adaptação do jQuery para o modelo de especificação de componentes. Para isso, foi necessário apenas a criação do arquivo descritor (component.json) e compactação em arquivo ZIP. A Figura 18 mostra a estrutura de arquivos e diretórios do componente.


**Figura 18: Estrutura de arquivos do componente jQuery.**

A Figura 18 apresenta a estrutura dos arquivos do componente jQuery de acordo com o modelo de especificação de componentes.

O segundo componente criado consiste, também, na adaptação do *Twitter Bootstrap* para o modelo de especificação de componentes. A escolha do *Twitter Bootstrap* deve-se a sua relação com o jQuery, pois, ao utilizar o arquivo Javascript do *Twitter Bootstrap*, deve-se, obrigatoriamente, utilizar o jQuery. Dessa forma pode-se, também, testar a dependência entre os componentes, que

nesse caso não ocorre todas as vezes. A Figura 19 apresenta a estrutura de arquivo do componente do *Twitter Bootstrap*.

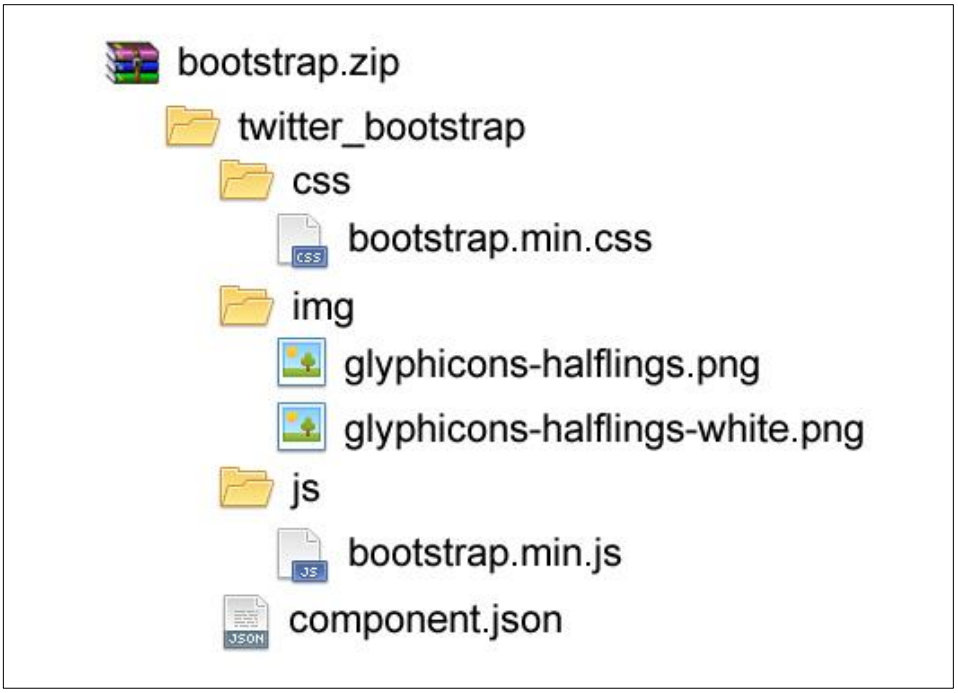


**Figura 19: Estrutura de arquivos do componente *Twitter Bootstrap*.**

Como mostra na Figura 19, a arquitetura do componente do *Twitter Boostrap* é composta por três diretórios além do diretório raiz. O diretório “css” contém os arquivos de estilo, o “img” contém as imagens e o “js” contém os arquivos Javascript.

Após a criação dos componentes de exemplo, realizou-se uma série de testes. Os testes foram divididos em duas etapas: teste de envio de componentes ao repositório; e teste utilização de componente pelo *middleware*. Para o envio de componentes, definiram-se os seguintes casos de teste:

- Envio de um componente compactado em um formato inválido, como RAR;
- Envio de um componente sem o arquivo descritor;
- Envio de um componente com o diretório raiz com o nome errado;
- Envio de um que contém dependência não cadastrada no repositório;
- Envio de um componente corretamente sem dependência;

- Envio de um componente corretamente com dependência;
- Envio de um componente já existente;
- Envio de uma nova versão de um componente;

Para a utilização dos componentes pelo *middleware*, definiram-se os seguintes testes:

- Utilização de um repositório inexistente;
- Utilização de um componente inexistente;
- Utilização de um componente válido;
- Utilização de um componente válido com dependência;

Os primeiros testes realizados foram de envio de componentes ao repositório. Portanto, verificou-se a necessidade de criar componentes inválidos de acordo com cada teste. A lista a seguir apresenta os resultados dos testes:

- **Envio de um componente compactado em um formato inválido:** o repositório exibiu a mensagem de erro “O arquivo deve estar no formato 'zip'”;
- **Envio de um componente sem o arquivo descritor:** o repositório exibiu a mensagem de erro “Não foi possível encontrar o arquivo 'component.json' dentro do zip”;
- **Envio de um componente com o diretório raiz com o nome errado:** o repositório exibiu a mensagem de erro “O nome do componente está diferente do nome diretório compactado”;
- **Envio de um que contém dependência não cadastrada no repositório:** o repositório exibiu a mensagem de erro “O componente 'jquery' não foi encontrado em nosso banco de dados”;
- **Envio de um componente corretamente sem dependência:** o repositório exibiu a mensagem de sucesso “Seu componente foi publicado”;
- **Envio de um componente corretamente com dependência:** o repositório exibiu a mensagem de sucesso “Seu componente foi publicado”;

- **Envio de um componente já existente:** o repositório exibiu a mensagem de erro “Já existe um componente com este nome e versão”;
- **Envio de uma nova versão de um componente:** o repositório exibiu a mensagem de sucesso “Seu componente foi publicado”.

Na segunda etapa dos testes, que consiste na utilização dos componentes por meio do *middleware*, criou-se, também, um *software* de exemplo para realização dos testes. A Figura 20 ilustra a estrutura de arquivos do *software* de teste.

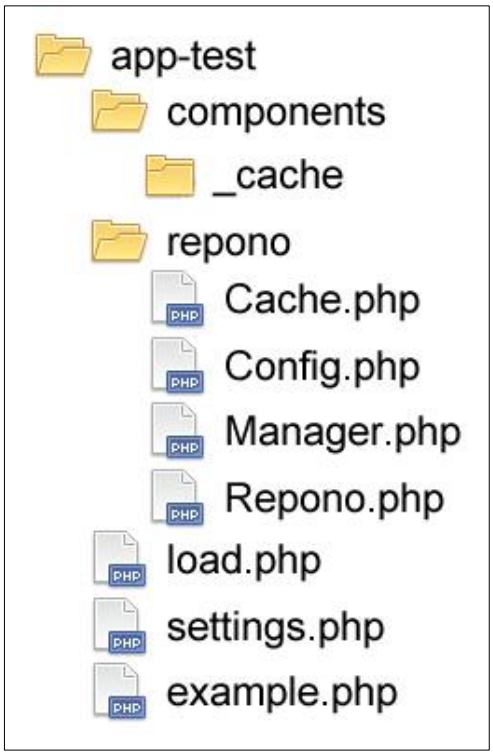


**Figura 20: Estrutura de arquivo do *software* de teste.**

A Figura 20 apresenta a estrutura de arquivos do *software* criado como teste. A estrutura pode ser entendida da seguinte maneira:

- **app-test:** diretório raiz do *software*;
    - **components:** repositório local de componentes;
        - **_cache:** diretório para guardar *cache* com informações sobre os componentes;

- **repono:** diretório com as classes do *middleware*;
- **load.php:** arquivo de inicialização do *software*. É responsável por importar os arquivos;
- **settings.php:** arquivo de configuração do *software*. Nele são definidas as configurações que são aplicadas ao *middleware*;
- **example.php:** arquivo com as chamadas dos componentes.

Os testes da segunda etapa foram executados de acordo com a sequência planejada. **Utilização de um repositório inexistente:** foi definido, no arquivo de configuração, a URL de um repositório inválido. No caso, utilizou-se a URL “http://repositorioinvalido.com”, como mostra a Listagem 9.

```
Config::set('url_repository', ' http://repositorioinvalido.com/');
```

**Listagem 9: Configuração da URL do repositório.**

Na utilização de um repositório inexistente, como apresentado na Listagem 9, o *middleware* disparou uma exceção com a mensagem “Repositório 'http://repositorioinvalido.com/' não encontrado”.

**Utilização de um componente inexistente:** foi utilizado o nome único de um componente inexistente no repositório para verificar o comportamento do *middleware*. A Listagem 10 apresenta o código utilizado para realização do teste.

```
<script src="<?= Repono::get('inexistente') ?>/file.js"></script>
```

**Listagem 10: Utilização de componente inexistente.**

Na utilização de um componente inexistente, como apresentado na Listagem 10, o *middleware* disparou uma exceção com a mensagem “O componente 'mootools' não foi encontrado”.

**Utilização de um componente válido:** foi utilizado um componente válido para verificar o comportamento do *middleware*. No caso, utilizou-se o componente jQuery, criado anteriormente. A Listagem 11 apresenta o código utilizado para realização do teste com um componente válido.

```
<script src="<?= Repono::get('jquery') ?>/jquery.js"></script>
```

**Listagem 11: Utilização do componente jQuery.**

A Listagem 11 apresenta o código de utilização de componente válido, pois está cadastrado no repositório. O *middleware* realizou o *download* do componente, jQuery, e extraiu os arquivos para o repositório local de componentes.

**Utilização de um componente válido com dependência:** foi utilizado um componente válido que contém dependências. No caso, foi utilizado o componente *Twitter Bootstrap*, criado anteriormente e que tem dependência do jQuery. Para realização desse teste, removeu-se o componente jQuery utilizado anteriormente. A Listagem 12 apresenta o código utilizado para realização do teste.

```
<script src="<?= Repono::get('bootstrap') ?>/
/js/bootstrap.min.js"></script>
```

**Listagem 12: Utilização do componente *Twitter Bootstrap*.**

No código apresentado na Listagem 12, o *middleware* realizou o *download* do componente *Twitter Bootstrap*, verificou as dependências e em seguida realizou o *download* do componente jQuery.

Com a realização dos testes pode-se observar que tanto o *Repono*, quanto o *middleware*, funcionaram de forma eficiente. Os resultados esperados foram obtidos com êxito, pois as falhas ocasionadas foram planejadas para acontecer, assim como o funcionamento correto em determinado momento.

# 5 CONSIDERAÇÕES FINAIS

Para a elaboração deste trabalho foi necessário realizar pesquisas em livros, monografias, dissertações, teses e artigos científicos. Com isso pôde-se aprender sobre os conceitos de Reuso de *Software*, Engenharia de Domínio e, principalmente, Engenharia de *Software* Baseada em Componentes.

Pôde-se aprender, também, sobre as várias técnicas de Reuso de *Software*, assim como sua granularidade, que vai desde o reuso de funções até *softwares* completos.

Na parte referente à Engenharia de *Software* Baseada em Componentes pôde-se entender o que são componente e modelo de especificação de componentes. Além disso, foi possível identificar a importância da utilização de um processo de desenvolvimento de *software* voltado para o reuso de componentes, pois as etapas do processo são diferentes de um processo comum. Isso porque, ao invés de implementar o *software* inteiro, deve-se buscar componentes existentes que satisfaçam determinado requisito.

Como resultado deste trabalho teve-se a criação do modelo de especificação de componentes voltado para o ambiente *web*. Esse modelo contém um conjunto de especificações para o desenvolvimento de um componente e de um *middleware,* além de um repositório de componentes. Outros trabalhos podem ser realizados com esse modelo de especificação, inclusive, para o desenvolvimento do *middleware* e/ou repositório para outras plataformas.

Os códigos-fonte dos projetos produzidos, em PHP, estão disponíveis de forma aberta (*open source*) sob a licença *BSD 3 – Clause License*. Os códigos-fonte estão disponíveis nos endereços:

- Repositório: https://bitbucket.org/vaneves/repono
- *Middleware*: https://bitbucket.org/vaneves/reponomiddleware
- Aplicação de Exemplo: https://bitbucket.org/vaneves/reponoexample

Uma instância do *Repono* está disponível para acesso, dessa forma, pode-se utilizá-la para envio e *download* de componentes. O repositório está disponível

no endereço http://repono.vaneves.com/. Os componentes de exemplo desenvolvidos neste trabalho estão disponíveis no repositório. Porém, as licenças dos componentes estão de acordo com seus arquivos originais.

Como trabalhos futuros, pretende-se escrever e publicar artigos sobre este trabalho. Além disso, pretende-se, também, continuar trabalhando para melhorar o modelo de especificação. Como uma das futuras melhorias, pretende-se fazer com que o *middleware* possa consultar em mais de um repositório ao mesmo tempo, verificando qual dele oferece uma versão melhor para o componente. Pretende-se, também, trabalhar no desenvolvimento do middleware para outras plataformas.

O modelo de especificação de componentes, criado neste trabalho, será aplicado na Fábrica de *Software* do CEULP/ULBRA. Pois há uma circulação grande de desenvolvedores nos projetos, tanto de funcionários, como de acadêmicos, que participam por meio de estágios, trabalhos de conclusão de cursos ou programas de iniciação científica. Dessa maneira, a utilização do modelo de especificação de componentes fará com que o desenvolvimento fique padronizado e reutilizável.

## 6 REFERÊNCIAS BIBLIOGRÁFICAS

BACON, J. **Practical PHP and MySQL:** Building Eight Dynamic Web Applications. Boston: Prentice Hall, 2007. 512 p.

BLOIS, A. P. B. **Uma abordagem de projeto arquitetural baseado em componentes no contexto de engenharia de domínio**. 2006. 207 p. Tese de Doutorado (Engenharia de Sistemas e Computação) - Coordenação dos Programas de Pós-Graduação de Engenharia, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

D'SOUZA, L. P., WILLS, A. C. **Objects, components, and frameworks with UML:** the Catalysis approach. California: Addison-Wesley, 1999. 785 p.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures.** 2000. PhD Dissertation - Information and Computer Science, University of California, Irvine.

GAMA, Erich et al. **Padrões de projeto:** soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.

GOMES, A. T. A. **CORBA para Computação Móvel**. 2001. 20 p. Trabalho de Conclusão de Curso (Introdução à Computação Móvel) - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro.

Introducing JSON. JSON. Disponível em <http://www.json.org>. Acesso em: 13 maio 2013.

Manual do PHP. PHP: Manual do PHP – Manual. Disponível em <http://www.php.net/manual/pt_BR/index.php>. Acesso em: 13 maio 2013.

NEVES JR, V. C. **Trilado:** um framework MVC desenvolvido em PHP. In: ENCOINFO, nº 12, 2010, Palmas. **Anais**... p. 123 – 132.

NUGET. Overview. Disponível em <http://docs.nuget.org/docs/start-here/overview>. Acesso em: 10 junho 2013.

OLIVEIRA, R. F. **Formalização e verificação de consistência na representação de variabilidades**. 2006. 133 p. Dissertação de Mestrado

(Engenharia de Sistemas e Computação) - Coordenação dos Programas de Pós-Graduação de Engenharia, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

OTTO, Mark. Bootstrap from Twitter. Disponível em <https://dev.twitter.com/blog/bootstrap-twitter>. Acesso em: 28 maio 2013.

PRESSMAN, Roger S. **Engenharia de software**. tradução José Carlos Barbosa dos Santos. São Paulo: Pearson Education, 1995. 1056 p.

PRESSMAN, Roger S. **Software Engineering:** A practitioner's Approach. 4. ed. New York : McGraw-Hill, 1997. 852 p.

PRESSMAN, Roger S. **Engenharia de Software**. 4. ed. New York : McGraw-Hill, 1997. 852 p.

RICCIONI, P. R**. Introdução a Objetos Distribuídos com CORBA.** Florianópolis: Visual Books, 2000. 104 p.

SAMETINGER, J. **Software Engineering with Reusable Components**. New York: Springer, 1997. 272 p.

SNELL, J., TIDWELL, D., KULCHENKO, P. **Programming Web Service with SOAP.** 1 ed. O'Reilly, 2002. 244 p.

SOMMERVILLE, Ian. **Engenharia de Software**. 8. ed. São Paulo: Pearson, 2010. 552 p.

VASCONCELOS, A. P. V. **Uma abordagem de apoio à criação de arquiteturas de referência de domínio baseada na análise de sistemas legados**. 2007. 267 p. Tese de Doutorado (Engenharia de Sistemas e Computação) - Coordenação dos Programas de Pós-Graduação de Engenharia, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

WEINREICH, R., SAMETINGER, J. Component Models and Component Services: Concepts and Principles, In: HINEMAN, G. T. e COUNCILL, W. T. (Eds.). **Component-Based Software Engineering:** Putting the Pieces Together. Addison-Wesley, 2001. p. 33-48.

XAVIER, Otávio C. **Serviços Web Semânticos Baseados em RESTful:** Um Estudo de Caso em Redes Sociais Online. 2011. 135 p. Dissertação de Pós-Graduação (Sistemas de Informação) - Programa de Pós–Graduação do Instituto de Informática, Universidade Federal de Goiás, Goiânia.

# ANEXOS

# ANEXO A – BSD 3-*Clause License*



Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the Valdirene da Cruz Neves Júnior nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.