UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
DIRETORIA DE PESQUISA E PÓS-GRADUAÇÃO
ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

THIAGO ALEXANDRE LENZ

# USO DE COMPONENTES NA PROGRAMAÇÃO: ESTUDO EXPERIMENTAL COM O FRAMEWORK OSGI

MONOGRAFIA DE ESPECIALIZAÇÃO

MEDIANEIRA
2011

THIAGO ALEXANDRE LENZ

# USO DE COMPONENTES NA PROGRAMAÇÃO: ESTUDO EXPERIMENTAL COM O FRAMEWORK OSGI

Monografia apresentada como requisito parcial à obtenção do título de Especialista na Pós Graduação em Engenharia de *Software*, da Universidade Tecnológica Federal do Paraná – UTFPR – Câmpus Medianeira.

Orientador: Prof Me. Fernando Schütz

MEDIANEIRA

2011

Ministério da Educação
**Universidade Tecnológica Federal do Paraná**
Diretoria de Graduação e Educação Profissional
Especialização em Engenharia de Software

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

## TERMO DE APROVAÇÃO

**Uso de componentes na programação: Estudo experimental com o *framework* OSGi**

Por

**Thiago Alexandre Lenz**

Esta monografia foi apresentada às 09:30 h do dia 10 de dezembro de 2011 como requisito parcial para a obtenção do título de Especialista no curso de Especialização em Engenharia de *Software*, da Universidade Tecnológica Federal do Paraná, Câmpus Medianeira. Os acadêmicos foram argüidos pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. Me. Fernando Schütz
UTFPR – Campus Medianeira
(orientador)

_____
Prof. M.Eng. Juliano Lamb
UTFPR – Campus Medianeira

_____
Prof. Me Alan Gavioli
UTFPR – Campus Medianeira

## AGRADECIMENTOS

Primeiramente agradeço a Deus por me dar saúde, pelo dom da vida, pela fé e perseverança para vencer os obstáculos.

Aos meus pais, irmãos e avós, pela orientação, dedicação e incentivo nessa fase do curso de pós-graduação e durante toda minha vida.

Agradeço aos amigos que eu fiz em todas as empresas em que trabalhei, e que de certa forma, me incentivaram a escolha do tema desse trabalho.

“O que empobrece o ser humano, não é a falta de dinheiro, mas sim, a falta de fé, motivação e criatividade.”

(ANA FRAGA)

## RESUMO

LENZ, Thiago Alexandre. Uso de componentes na programação: Estudo experimental com o Framework OSGi. 2011. 92. Monografia (Especialização em Engenharia de Software). Universidade Tecnológica Federal do Paraná, Medianeira, 2011.

Existe uma série de dificuldades na modularização de aplicativos. Tais dificuldades são encontradas tanto no levantamento das funcionalidades como na própria construção, bem como no projeto de um sistema para que seus módulos sejam de baixo acoplamento, tenham alta coesão, sejam dinâmicos e flexíveis. Este trabalho discute o desenvolvimento de *software* baseado em componentes, através de um levantamento bibliográfico sobre como definir módulos e componentes na percepção do sistema bem como, sua representação em diagramas da UML. O trabalho também aborda questões sobre a arquitetura do *framework OSGi*, que possui três níveis de utilização: Modularização, Ciclo de vida e Serviços. Para cada nível é desenvolvido um protótipo para validar as suas funcionalidades. Em cima desses protótipos são apresentados os scripts de testes aplicados.

**Palavras-chave:** Coesão. Dinamismo. Flexibilidade. Modularização. Baixo acoplamento.

## ABSTRACT

LENZ, Thiago Alexandre. C Uso de componentes na programação: Estudo experimental com o Framework OSGi. 2011. 92. Monografia (Especialização em Engenharia de Software). Universidade Tecnológica Federal do Paraná, Medianeira, 2011.

There are a number of difficulties in the modularization of applications. Such difficulties are found in both the survey of features such as the construction itself, as well as in designing a system so that its modules are loosely coupled, have high cohesion, be dynamic and flexible. This paper discusses the development of component-based software, through a literature review on how to define modules and components in the perception of the system and its representation in UML diagrams. The paper also addresses questions about the architecture of the OSGi framework, which has three levels of use: Modularization, and life cycle services. For each level is a prototype to validate the functionality. On top of these prototypes are presented scripts tests.

**Keywords:** High Cohesion. Dynamics. Flexibility. Modularization. Loosely Coupled.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE QUADROS

## LISTA DE ABREVIATURAS

| | |
|---|---|
| AD | Análise de Domínio |
| API | *Application Programming Interface* |
| DBC | Desenvolvimento Baseado em Componentes |
| ED | Engenharia de Domínio |
| ES | Engenharia de Software |
| ESBC | Engenharia de Software Baseada em Componentes |
| IDE | *Integrated Development Environment* |
| Java SE | *Java Standard Edition* |
| JVM | *Java Virtual Machine* |
| LCD | *Liquid Crystal Display* |
| NFe | Nota Fiscal Eletrônica |
| OSGi | *Open Service Gateway initiative* |
| RH | Recursos Humanos |
| UML | Unified Model Language |
| URL | *Uniform Resource Locator* |

**SUMÁRIO**

# 1 INTRODUÇÃO

Quando se projeta a construção de uma casa, é possível escolher se ela será de alvenaria ou de madeira, se as portas e janelas serão de madeira ou de alumínio, se ela será branca ou azul, se a televisão será de plasma ou LCD (*Liquid Crystal Display*), enfim, cada espaço pode ser escolhido, de acordo com as preferências do cliente. Caso houver algum problema com algum desses itens, ou houver uma troca, não existe empecilho para isso. Às vezes o orçamento é curto, deixam-se algumas coisas para projetos futuros, os chamados puxados. Esses entre muitos outros são os componentes de uma casa, cada parte tem um propósito especifico e bem definido.

Desenvolver *software* pode se tornar muito semelhante, pode se escolher o tipo de banco de dados, optar por *hibernate* ou *toplink*, por exemplo, e quanto à interface, será *Web* ou *desktop*, e quando se precisa realizar uma customização para um cliente (comparando com o puxado da casa), essas são questões arquiteturais que podem ser trocadas a qualquer momento no ciclo de vida do *software*.

No âmbito de negócio também existem considerações. É possível separar os sistemas em módulos, como: financeiro, contábil, faturamento, recursos humanos, etc. Cada módulo pode ou não ser independente, por exemplo, o módulo faturamento pode gerar movimento contábil, mas ele não precisa se preocupar se o movimento gerado será enviado ao mesmo sistema ou via integração a outro. Isso gera flexibilidade, pois uma empresa pode trabalhar com vários sistemas integrados, e quando desejar-se migrar de um módulo especifico para outro, sabe-se que o restante se comportará normalmente, pois apenas será trocado um componente de *software* por outro.

Uma das grandes problemáticas na ES (Engenharia de *Software*) consiste em como aperfeiçoar os projetos de *software*. As preocupações são grandes, pois desenvolver *software* envolve custos monetários e temporais, e as demandas estão cada vez maiores. Segundo Hall *et al.* (2010), uma das estratégias adotadas pela comunidade de ES nos últimos anos tem sido a reutilização, representando uma importante solução para muitas dificuldades no desenvolvimento de *software*.

Os maiores benefícios da reutilização de *software* se referem ao aumento da qualidade e redução do esforço de desenvolvimento. Com esse conceito de reutilização de *software*, por meados de 1968, surgem os componentes de *software*, cuja ideia é decompô-los em unidades reutilizáveis. Uma vez divididos em componentes/módulos, estes podem ser reutilizadas em

outros projetos, como por exemplo, um componente com a finalidade de realizar cálculos matemáticos complexos, poderia ser utilizado em mais de um projeto da mesma empresa, ou até mesmo, ser vendido a terceiros (GIMENEZ e HUZITA, 2005).

Setores da indústria já possuíam a ideia de componentização para aperfeiçoar a produção de bens e serviços. Tem-se, como exemplo, a indústria de componentes elétricos que são conectados por fios para se tornar componentes maiores. A ideia de componentes de *software* é similar a estas, pois a interligação entre vários componentes faz que os mesmos se tornem um único produto (GIMENEZ e HUZITA, 2005).

O conceito de OO (Orientação a Objetos) veio como promessa de ser um mecanismo desenvolvimento de *software* baseado em componentes, porém não conseguiu o sucesso que almejava segundo Gimenez e Huzita (2005), pois os fatores afirmam que ele fornece muitos recursos para a produção de componentes, mas não alavancou a esperada reutilização em massa. O fato é que Java é uma das principais linguagens desse conceito, e a mesma não se preocupou em oferecer suporte explicito para modularização, apenas ao encapsulamento de dados no modelo de objetos, (HALL *et al.* 2010).

Em meados de 1999 surgiu uma aliança chamada OSGi (*Open Service Gateway initiative*), cuja finalidade é atender essa falta de apoio a modularização em Java através da proposta de um novo modelo de programação orientada a serviços (HALL *et al.* 2010). Essa tecnologia é um conjunto de especificações que definem um sistema de componentes dinâmicos (The OSGi Architecture, 2011). Através dessa especificação surgem algumas implementações de código aberto como: *Apache Felix, EquinoX* e o *Knopflerfish* (GÉDÉON, 2010) .

Este trabalho apresenta a aplicação dos conceitos de componentização utilizando o OSGi como *framework* de desenvolvimento.

## 1.1 OBJETIVO GERAL

Desenvolver um estudo experimental sobre aplicativos de baixo acoplamento utilizando modularização em Java *Standard Edition* (Java SE) e componentes de *Software* desenvolvidos com base na especificação do *framework* OSGi.

## 1.2 OBJETIVOS ESPECÍFICOS

- Desenvolver um referencial teórico sobre componentes de *software*, processos de engenharia e sobre a arquitetura do *framework* OSGi;
- Desenvolver um protótipo para cada um dos três níveis de utilização do *framework* OSGi (*modularization*, *lifecycle* e *services*) através da implementação *Apache Felix*.
- Apresentar resultados de testes dos protótipos

## 1.3 JUSTIFICATIVA

O principal objetivo do DBC (Desenvolvimento Baseado em Componentes) é a reutilização dos mesmos em outros projetos ou produtos. Outro ponto importante é a flexibilidade, uma vez que os *softwares* estão cada vez mais abrangentes e complexos, quebrar esses grandes problemas em partes menores tende a facilitar o desenvolvimento e manutenção como um todo. Segundo IEEE (1990), é considerado flexível o sistema ou componente que possui a facilidade de ser modificado para ser usado em outras aplicações além daquela que ele foi projetado inicialmente.

As linguagens orientadas a objetos, especificamente Java, fornecem ótimos conceitos como encapsulamento e polimorfismo, mas somente isso não evita problemas que podem ocorrer em tempo de execução. Quando se executa uma aplicação em Java, várias são as classes a serem carregadas para a JVM (*Java Virtual Machine*) onde passam pela geração do *bytecode*. É comum haver vários arquivos *.jar*, é comum também haver diferentes versões da mesma classe, porém em *jars* distintos. Para Hall *et al.* (2010), o *classpath* não presta atenção nas versões de código, ele simplesmente retorna à primeira versão que ele encontra. Muitos erros podem ocorrer em tempo de execução, como por exemplo: métodos não encontrados (*NoSuchMethodError*), falha ao converter tipos (*ClassCastException*) ou ainda erros lógicos devido a alteração do algoritmo de um método.

Quando se fala em quebrar um *software* em componentes, apenas quebrar em arquivos *jars* não é suficiente, no exemplo citado acima não há um mecanismo adequado para determinar qual versão de uma classe deva-se usar. Não é possível realizar a instalação de um novo componente sem parar toda a aplicação. Não há garantias que esse novo componente ou atualização de um existente não venha a comprometer a estabilidade da aplicação, podendo

haver um terceiro componente requerido que não foi configurado no *classpath*. O OSGi propõe-se a resolver esses problemas fornecendo um mecanismo de gerenciamento de componentes.

Hoje é muito comum haver a necessidade de sistemas de alta disponibilidade. Conforme DELPIZZO (2008), entende-se por sistema de alta disponibilidade aquele resistente a falhas de software, de hardware e de energia. É considerado seguro e confiável um sistema capaz de se manter no ar mesmo após essas falhas. Dividir o *software* em partes onde as mesmas são disponibilizadas em serviços tende a trazer uma série de vantagens: facilitar a manutenção; prover baixo acoplamento, consumidores não tem conhecimento dos provedores; e ser dinâmica, a troca da utilização de serviços em tempo de execução. Quando for realizada uma atualização ou instalação de um novo serviço, não pode se parar os consumidores, quando um serviço está em manutenção, outro deverá ser colocado em seu lugar, ou então apenas removê-lo quando for instalada a nova versão.

## 1.4 LIMITAÇÃO DE ESTUDO

O escopo de estudo se limita às características básicas do OSGi *framework* para as aplicações Java versão Java SE (Java *Standard Edition*).

## 1.5 ESTRUTURA DO TRABALHO

O presente trabalho está estruturado em 5 capítulos, sendo no primeiro capítulo a introdução, objetivos e justificativa.

No capítulo 2, seção 2.1, é apresentado o conceito de componente de *software* bem como sua representação na UML. Na sequência a seção 2.2 é descreve como identificar tais componentes no momento da percepção do sistema utilizando o processo da Engenharia de Domínio (ED). Já na seção 2.2 é apresentada a arquitetura do *framework* do OSGi, são descritas as soluções propostas para os problemas reais das aplicações Java. Na seção 2.3 é descrito sobre a arquitetura do *framework* OSGi, é realizada uma explicação sobre os

comportamentos de cada nível. Na seção 3 são apresentados os protótipos para cada um dos níveis, bem como os resultados dos testes de validação.

No capítulo 3 é apresentado o desenvolvimento de protótipos utilizando os conceitos levantados no capítulo 2. Sendo considerado um protótipo para cada nível: seção 3.4 modularização, seção 3.5 protótipo de ciclo de vida e na seção 3.6 o protótipo de serviços.

Já no capitulo 4 é exibido os resultados de testes aplicados nesses componentes desenvolvidos no capítulo 3.

E no capítulo 5 apresentado as considerações finais e trabalhos futuros.

# 2 REVISÃO BIBLIOGRÁFICA

Esse capítulo tem como objetivo apresentar a visão de vários autores sobre o processo de desenvolvimento de *software*, processos de desenvolvimento através da modularização utilizando componentes. É apresentado também a conceituação sobre o *framework* OSGi, detalhando alguns problemas da linguagem Java e do conceito de Orientação a Objetos, bem como as soluções que o *framework* propõe para tais problemas.

## 2.1 COMPONENTES DE SOFTWARE

Além de proporcionar a reutilização de *software*, o desenvolvimento baseado em componentes vem mudando significativamente o desenvolvimento de *software*. Conforme Resende (2007), além de diminuir os custos pode propiciar uma produção mais rápida e confiável de *software*, através do uso de componentes bem especificados e testados.

Um componente é uma parte do sistema com propósito especifico, pode ser desde um simples registrador de *log* a um módulo complexo. Não existe uma definição para o termo "componente de *software*". Para Vilella (2000), cada grupo de pesquisa caracteriza a maneira mais adequada ao seu contexto o que seria um componente e, assim, não há ainda na literatura uma definição comum para esse termo. O que se encontra nas literaturas são consensos das características que os componentes devem apresentar. Para Gimenez e Huzita (2005) um componente de *software* pode ser definido como uma unidade de *software* independente, que encapsula seu projeto e implementação e oferece interfaces bem definidas para o meio externo; todos os componentes devem oferecer interfaces[1], pois através delas que serão realizadas as interconexões entre os componentes do sistema, tornando-se assim um sistema baseado em componentes.

[1] Interface: classe abstrata que define atributos e métodos.

Gimenez e Huzita (2005) sugerem que o desenvolvimento e especificação de componentes, devem atender os seguintes requisitos:

- Um componente deve fornecer uma especificação clara de seus serviços, as interfaces devem ser identificadas e disponibilizadas separadamente, na qual o componente apenas realiza a assinatura. O componente deve se comportar de acordo com os parâmetros passados através dessa interface;
- As interfaces requeridas também devem ser definidas explicitamente. Essas interfaces declaram que necessitam de outros componentes para cumprir o seu propósito;
- A integração entre componentes deve ser exclusivamente através das interfaces.

Essas interfaces podem ser de dois tipos: **interfaces requeridas** e **interfaces fornecidas**. A interface requerida define entradas de funcionalidades requeridas, já a interface fornecida expõe funcionalidades na qual o componente se propõe a fazer. Conforme se observa na **Figura 1**, um componente (*ComponenteY*) deve oferecer uma interface (`InterfaceFornecida`) para que os demais componentes possam se conectar a ele, e pode conter interfaces requeridas (`InterfaceRequerida`), através do qual outros componentes serão conectados a ele. Conforme Gimenez e Huzita (2005), se um componente não possuir interfaces requeridas ele é um componente autocontido, ou seja, independente.

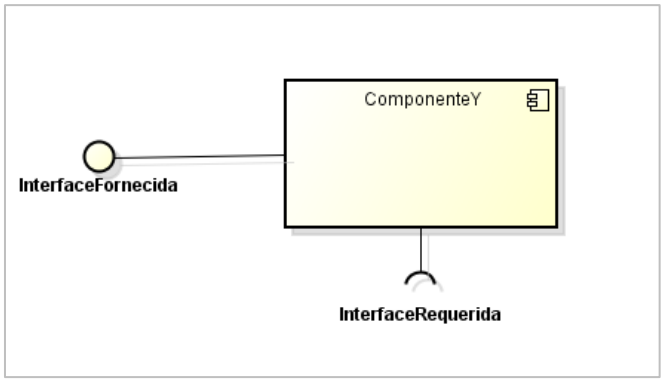



**Figura 1 – Componente de *software* com suas interfaces**

Na sequência a **Figura 2** apresenta um novo componente (*ComponenteX*) no qual está ligado ao *ComponenteY*, a conexão é realizada através da interface (`InterfaceRequerida`).

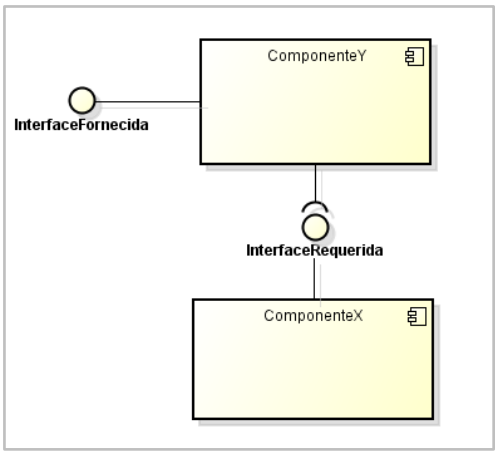



**Figura 2 – Conexão entre dois componentes**

### 2.1.1 Substituição, manutenção e reutilização de componentes

Conforme Gisele e Huzita (2005), componentes podem ser interconectados para formar componentes maiores. Segundo Barbosa e Perkusish (2007), O DBC facilita a manutenção dos sistemas por possibilitar que eles sejam atualizados pela integração de novos componentes e/ou substituição dos componentes existentes. Na **Figura 3** pode-se observar a configuração de um sistema baseado em vários componentes.

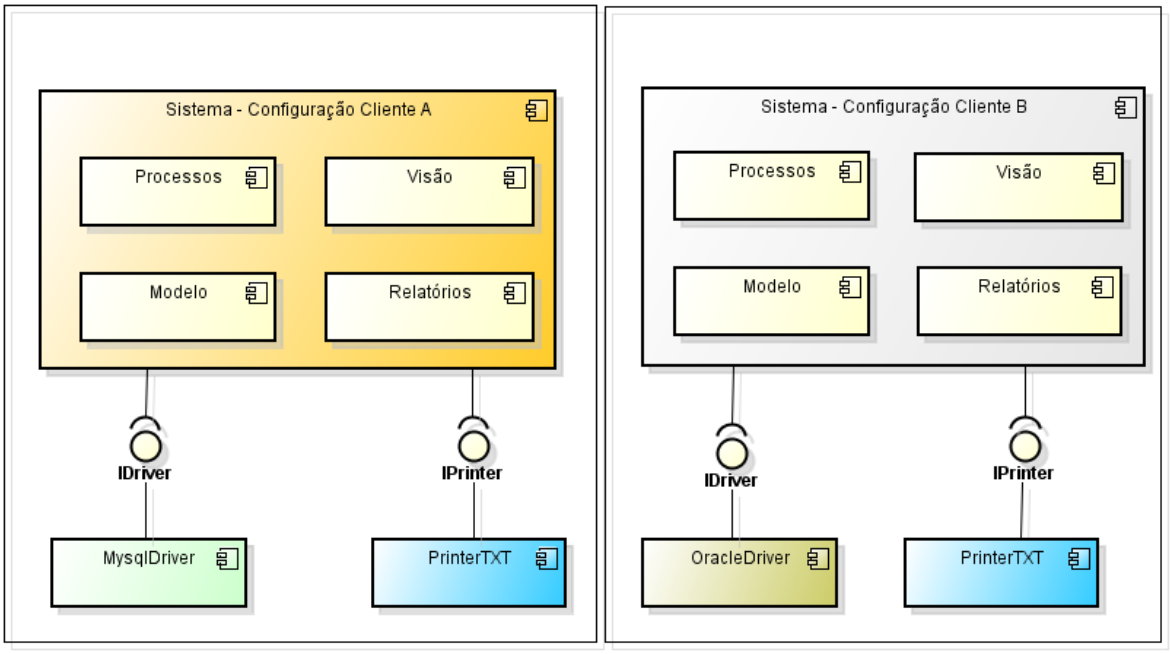



**Figura 3 – Complexo de Componentes com reutilização e substituição**

Nota-se na **Figura 3** que existem duas configurações semelhantes, através dessas duas configurações aplicou-se o conceito da reutilização e da substituição. O Conceito da substituição pode ser percebido através da interface requerida (`IDriver`), onde em uma configuração é utilizado `MysqlDriver` e `OracleDriver` na outra. Já o conceito de reutilização pode ser percebido no componente na outra interface requerida (`IPrinter`), onde foi utilizado o mesmo componente (*PrinterTXT*) nas duas configurações.

O conceito de manutenção nesse caso pode ser realizado pontualmente, uma vez que os componentes são considerados como unidades e havendo interfaces bem definidas, uma simples correção ou melhoria interna não impactaria diretamente no restante dos componentes.

### 2.1.2 Componentes na UML

A UML (*Unified Model Language*) atualmente em sua versão 2.0 possui no grupo de diagramas estruturais o diagrama de componentes, diagrama no qual é apresentado nas **Figuras**: **1**, **2**, **3** e **4** as quais possuem algumas definições de como representar componentes de *software* na UML:

- Na Linha 1 tem-se apenas uma figura no formato de uma caixa representando um componente;
- Na linha 2 é representado um componente juntamente com sua interface fornecida;
- Já na linha 3 é semelhante a 2ª linha, porém apresenta uma interface requerida;
- Nas linhas 4 e 5 estão representadas duas formas de conexão entre componentes, uma no utilizando uma flecha (linha 5) e outra no formato de um meio circulo (linha4).

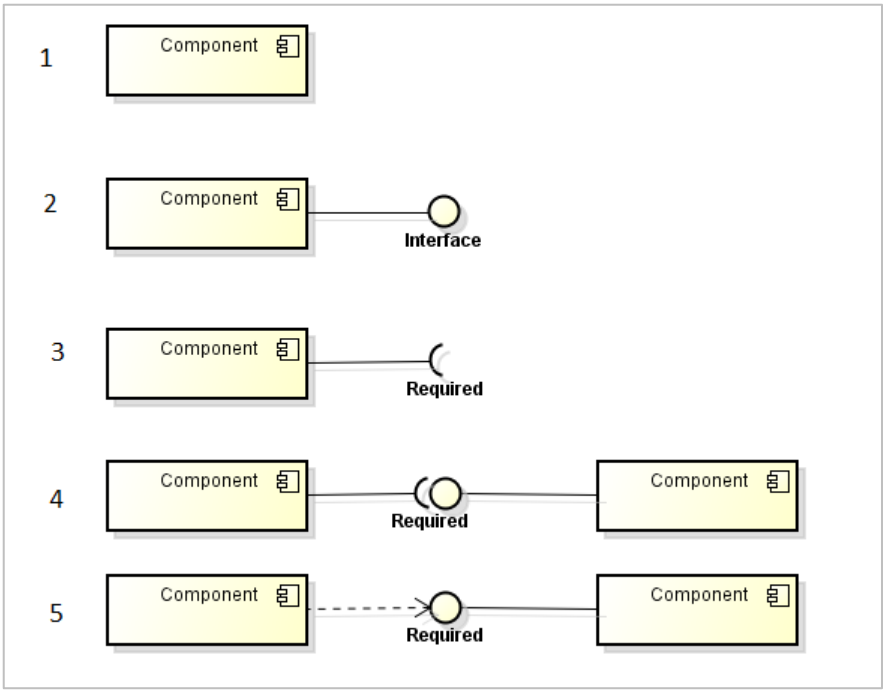



**Figura 4 – Representação de Componentes na UML**

### 2.1.3 Componentes e orientação a objetos

Pode-se considerar o conceito de componentes como uma evolução da orientação a objetos. A mesma oferece uma boa estrutura na questão de especialização e herança, no entanto à medida que o sistema cresce, essas heranças acabam se tornando pesadas e complexas, tornando o sistema inchado e difícil de localizar-se no meio do código, por mais documentado que seja.

Para Villela (2000), o DBC (Desenvolvimento Baseado em componentes) evoluiu da OO no sentido de restringir essas hierarquias no contexto do componente, adotando fortemente o conceito de agregação e associação ao invés de herança. Interconexões entre componentes através da agregação e associação garantem que seja utilizado apenas o que for restritamente necessário, sem haver a necessidade de levar toda a carga de código para todos os descendentes de uma classe. Conforme Heineman (1999) há a herança essencial onde são herdados comportamentos ou características visíveis e a herança acidental, onde se herda parte ou toda implementação de um código genérico para fins de reutilização, onde boa parte não é utilizada. Com essa definição pode-se dizer que herança do tipo acidental leva a um projeto pobre

## 2.2 PROCESSO DE IDENTIFICAÇÃO DE COMPONENTES

No item **Componentes de *Software*** foram apresentados os objetivos dos componentes de *software*, o qual não demonstra ser muito difícil de entender, porém, uma das maiores dificuldades é compreender quais elementos de um *software* podem vir a se tornar um componente reutilizável. Segundo Pietro-Diaz e Arango (1991), uma das maiores dificuldades na reutilização de *software* é, justamente, a criação de componentes que possam ser reutilizados em outras aplicações.

Esta seção tem por objetivo descrever o processo de ESBC (Engenharia de *Software* Baseado em Componentes), no qual está dividido em dois conceitos que de certa forma podem ser combinados para obter ótimos resultados: A Engenharia de Domínio (ED) e o Desenvolvimento Baseado em Componentes (DBC).

A Engenharia de Domínio possui foco em analisar e projetar componentes reutilizáveis com base nos projetos e produtos já existentes e nos possíveis a surgirem. Segundo Prietro-Diaz e Arango (1991), “a Engenharia de Domínio é o processo de identificação e organização do conhecimento sobre uma classe de problemas, o domínio do problema, para suportar sua descrição e solução”. Já o Desenvolvimento Baseado em Componentes visa ao máximo à combinação e adaptação de componentes já construídos,

conforme Gimenez e Huzita (2005) o DBC tem como principal objetivo o desenvolvimento de aplicações pela composição de partes já existentes.

### 2.2.1 Engenharia de domínio (ED)

Segundo Blois *et al*. (2004), existem muitos domínios que possuem características propagáveis em quase todas as suas aplicações, podendo fazer reuso dos mesmos processos e artefatos, o que promove a reutilização dos conceitos e funcionalidades em comum. Entende-se por domínio uma série de aplicativos com características comuns em si, como por exemplo: RH (Recursos Humanos), logística, manufatura, comercial, etc.

Durante o processo de análise de requisitos de um *software* muitos são os problemas que dificultam a organização dos mesmos para que possam ser reutilizados. Conforme Gimenez e Huzita (2005) os problemas que se mais destacam são:

- **Desconhecimento do domínio da aplicação**: na maioria dos casos não há conhecimento do negócio, todas as áreas estão demandando *software*, conhecer sobre todas é humanamente impossível. Em fator disso um pequeno tempo é destinado à equipe para conhecimento do domínio da aplicação;
- **Vocabulário conflitante**: grandes projetos requerem a integração de profissionais de diferentes áreas, e por vezes os termos utilizados por cada parte são conflitantes;
- **Incerteza nas decisões**: profissional trabalhando fora da área de especialização não possui certezas quanto ao que conhece;
- **Conhecimento errôneo do domínio da aplicação**: casos onde os desenvolvedores possuem informações errôneas sobre o funcionamento de alguns componentes

Segundo Blois *et al*. (2004) aplicações de um mesmo domínio possuem muitas características comuns e que, portanto, suas aplicações poderiam ser construídas a partir de um mesmo processo, conjunto de artefatos, e com isso, promovendo a reutilização dos conceitos e funcionalidades em comum.

A Engenharia de Domínio está dividida basicamente em três etapas: Análise de Domínio, Projeto do Domínio e Implementação do Domínio.

#### 2.2.1.1 Análise de Domínio (AD)

Na análise de domínio se determina os requisitos comuns de um grupo de sistemas do mesmo domínio que possam ser reutilizados. Segundo Gimenez e Huzita (2005), existem duas perspectivas de domínio: a primeira como uma coleção de problemas reais e, a segunda, domínio como uma coleção de aplicações (existentes e futuras) que compartilhem características em comum.

Para Neighbors (1981), a Análise de Domínio é uma tentativa de identificar os objetos, operações e relações, onde peritos em um determinado domínio percebem como importante. Considerando essa definição, em um domínio de RH existem objetos típicos como: Cartão Ponto, Folha de Pagamento, Calendário de Férias, Funcionário, Filial e Impressão de Olerite. Já como operação é possível citar: registrar batidas do cartão ponto, realizar o cálculo da folha de pagamento, solicitar e aprovar férias, admissão e demissão, etc. Considerando um domínio de um Hospital, se tem os seguintes objetos: Agenda de Consulta, Ficha de Consulta, Convênio, Tratamento, Médico e Unidade de Atendimento. Já as operações possíveis são: agendar consultas, realizar consulta, gerenciar convênios, etc.

Para Villela (2000), na fase de Análise de Domínio devem ser disponibilizadas representações que capturem o contexto e a abrangência do domínio, explicitando seu interfaceamento com outros domínios. A **Figura 5** representa requisitos explicados no parágrafo anterior, divididos cada um em seu respectivo domínio.


**Figura 5 – Requisitos de Domínios**

#### 2.2.1.2 Projeto do Domínio

A fase de projeto vem após a fase de análise de domínio. Nessa fase são refinadas as oportunidades de reutilização de componentes com propósito de refinar as especificações de projeto. Conforme Blois (2004), o projeto de domínio é a etapa em que os modelos de projetos são construídos, com base nos modelos de análise, no conhecimento obtido por estudos a respeito de projeto reutilizável e arquiteturas genéricas.

Para Villela (2000), as representações geradas na fase de projeto devem prever modelos arquiteturais que auxiliem na especificação de arquiteturas específicas para cada aplicação.

#### 2.2.1.3 Implementação do Domínio

Conforme Gimenez e Huzita (2005), a fase de implementação de domínio transforma as oportunidades de reutilização e soluções em um modelo de implementação, que inclui serviços tais como: a identificação, reengenharia e manutenção de componentes reutilizáveis que suportem os requisitos de projeto. Segundo Vilella (2000), os componentes implementacionais devem estar em conformidade com o modelo da fase de análise e a fase de projeto, de forma que os mesmos possam cooperar entre si para atender os requisitos da aplicação.

Nessa fase é realizada a escolha da tecnologia a ser utilizada. Segundo Blois (2004), essa fase é composta de duas atividades: Transformação de Modelos para uma Tecnologia Específica e Geração do Código.

#### 2.2.1.4 Profissionais envolvidos

Conforme Simos (1996) existem basicamente três profissionais envolvidos no processo de engenharia de domínio:

- Fontes: os usuários finais das aplicações existentes daquele domínio; especialistas do domínio que fornecem informações importantes relativas ao domínio;

- Produtores: analistas e projetistas do domínio que fazem a engenharia reversa das aplicações existentes, tendo como objetivo gerar análise, projeto e implementação e componentes reutilizáveis;
- Consumidores: são os desenvolvedores de aplicações e usuários finais interessados no entendimento do domínio, que utilizam modelos disponibilizados nas etapas da engenharia de domínio para a especificação de aplicações e também para obter um maior entendimento sobre conceitos e funções inerentes ao domínio.

### 2.2.2 Desenvolvimento Baseado em Componentes

Uma abordagem que estava sendo seguida há algum tempo é o desenvolvimento baseado em blocos monolíticos, os quais estão implicitamente interligados entre si, ou seja, com forte acoplamento e dependência. O DBC surgiu com o objetivo de quebrar esses blocos em componentes interoperáveis, reduzindo tanto a complexidade no desenvolvimento quanto os custos (Samentinger, 1997). Para Gimenez e Huzita (2005), o DBC tem como principal objetivo o desenvolvimento de aplicações pela composição de partes já existentes.

Uma vez que a ED possui foco no entendimento do domínio e como disponibilizar componentes reutilizáveis, o DBC está voltado a combinar vários componentes no desenvolvimento de uma aplicação. O DBC é auxiliado por metodologias e ferramentas. Para Brown e Wallnau (1998), uma ferramenta automatizada de DBC deve apresentar as seguintes características:

- Repositório de componentes, com intuito de armazenar os componentes utilizados no processo;
- Ferramenta de modelagem, na qual permita definir a descrição dos componentes bem como suas interações;
- Um gerador de aplicações baseadas em componentes;
- Um mecanismo de busca e seleção de componentes.

### 2.2.3 Métodos de Engenharia

Nessa seção é apresentada uma breve descrição sobre os principais métodos que auxiliam a ED e DBC. Conforme Vilella (2000), os métodos disponibilizam um modelo de domínio onde as abstrações de mais alto nível no domínio, representando principalmente os conceitos importantes do domínio, são modeladas. Pode se destacar os seguintes métodos de engenharia (Gimenez e Huzita, 2005):

- *Feature Oriented Domain Analysis* (FODA): captura as funcionalidades relacionadas ao domínio, auxiliando a identificação dos componentes que disponibilizariam as funcionalidades do mesmo;
- *Organization Domain Modelling* (ODM): e considerado um *framework* para criação de métodos de engenharia de domínio particularizados, conforme necessidades das organizações;
- *Evolutionary Domain Lyfe Cycle* (EDLC): adota uma abordagem gerativa de aplicações em um dado domínio, em contraste com a abordagem de componentes adotadas pelo FODA;
- *Feature-Oriental Reuse Method* (FORM): evolução do FODA, sendo mais voltado para o detalhamento de como desenvolver componentes a partir de funcionalidades (*features*) identificadas. Utiliza técnicas como *templates*, geração automática de código, parametrização, entre outros, para o desenvolvimento dos componentes;
- *Reuse-Driven Software Engineering Business* (RSEB): metodologia que amplia a abordagem de Orientação a Objetos, fornecendo pontos de variação, ou seja, partes específicas que podem acoplar novas construções que modifiquem ou customizem as funcionalidade de uma construção. Um ponto fraco é que não possui suporte automatizado para sua utilização, ou seja, uma ferramenta.
- *Featuring RSEB* (*FeatuRSEB*): uma evolução do RSEB através da tentativa de unir as vantagens do FODA, como a modelagem de abstrações de domínio;
- *Catalysis*: é um modelo de DBC detalhado, cobrindo todas as fases do desenvolvimento de um componente a ser utilizado no desenvolvimento de uma dada aplicação, desde a especificação até sua implementação (D'Souza e Wills, 1998);

- *KobrA*: utiliza o conceito de *framework* genérico e reutilizável, que é utilizado para instanciar aplicações. Cada um desses *frameworks* oferece uma descrição genérica sobre todos os elementos de *software* que uma família de aplicações deve possuir, incluindo as partes que variam entre elas;
- UML *Componentes*: considera componente como uma visão arquitetural da análise realizada sobre o problema. Esses componentes adotam princípios de orientação a objetos como: unificação de dados e funções, encapsulamento e identidade.

## 2.3 OSGI FRAMEWORK

Fundado pela OSGi Aliance no ano de 1999, o *framework* foi inicialmente planejado para ser utilizado em aplicações embarcadas em Java. Evoluiu com as especificações seguintes, estando atualmente na versão 4.0, onde se encaixa em qualquer aplicação em Java, e em qualquer dispositivo que possua suporte a JVM (GÉDÉON, 2010).

Conforme HALL *et al.* (2010), o *framework* provê um novo conceito de modularização em Java, o modelo de programação orientada a serviços, sendo conhecido como serviços na máquina virtual (*SOA in a VM*). Para GÉDÉON (2010), o OSGi provê um meio padronizado de execução e implantação de um sistema modular Java.

O *framework* define três níveis de funcionalidades, onde cada nível depende dos níveis mais básicos, conforme pode ser observado na **Figura 6** (HALL *et al.*, 2010):

- *Module Layer*: nível mais básico que oferece um nível de abstração que consiste apenas na separação de módulos em arquivos *jar*;
- *Lifecycle Layer*: nível intermediário, onde se tem um gerenciamento e instalação dos módulos em tempo de execução;
- *Services Layer*: nível avançado que dispõe de mecanismos para registro e pesquisa de serviços dinamicamente.

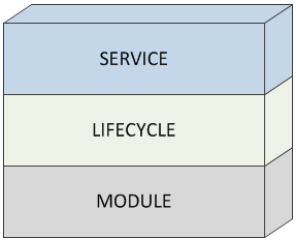



**Figura 6 – Camadas de funcionalidades do OSGi**

**Fonte: HALL *et al.*(2010)**

### 2.3.1 Modularização

O OSGi prove um mecanismo de modularização em Java. Mas afinal o que é modularização? Na área de desenvolvimentos de sistemas, é onde o código está dividido logicamente, onde cada parte atende uma necessidade especifica (HALL *et al.*, 2010). A **Figura 7** apresenta um exemplo de modularização, onde se tem quatro módulos com as suas respectivas dependências.

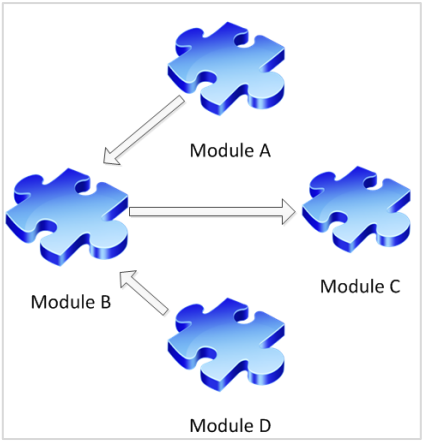



**Figura 7 – Sistema decomposto em módulos**

**Fonte: HALL *et al,* (2010)**

Considerando a **Figura 7** pode se observar que o módulo A depende do módulo B, o mesmo ocorre com o módulo D, já o módulo B depende do módulo C. Este é apenas um exemplo de como um sistema pode ser dividido em módulos. Esse conceito possui vantagens

tanto na teoria como na prática, na teoria fica mais fácil de compreender, na prática além da organização de código provê flexibilidade de adaptação e mudanças.

Os módulos tem a finalidade de definir fronteiras internas de uma aplicação. Existem basicamente dois tipos de módulos: físico e lógico. A modularidade física determina como os arquivos serão empacotados, quais arquivos irão pertencer a determinado *jar*. Já a modularidade lógica define as funcionalidades, responsabilidades e dependências de cada módulo. São dois conceitos diferentes que são facilmente confundidos. Pode-se existir modularização lógica sem modularização física, pois nenhuma aplicação é obrigada a quebrar seus módulos em arquivos separados. A modularização física apenas confirma e garante a modularização lógica.

#### 2.3.1.1 Modularização e Orientação a Objetos

A modularização possui o conceito de dividir um grande problema em problemas menores nos quais se relacionam entre si, esses problemas são definidos como componentes, onde estes devem oferecer uma interface de acesso às funcionalidades internas que são escondidas dos demais componentes. A programação orientada a objetos também divide os problemas em partes, denominando essas partes como classes, onde através da herança estas herdam as funcionalidades de uma classe mais abstrata.

Conforme pode ser observado na **Figura 8,** no lado esquerdo é representado um diagrama de classes onde se tem a classe `Car` que possui um objeto do tipo `Engine`. Há também 2 classes que especializam `Car`: `Ferrari` e `Golf`. No lado direito é representado um diagrama de componentes, onde se tem um componente `Engine` que provê a interface `EngineInterface` na qual os componentes `Ferrari` e `Golf` fazem conexão. `Ferrari` e `Golf` provem a interface `CarInterface`. A diferença básica entre os dois é que no diagrama de componentes não existe herança.

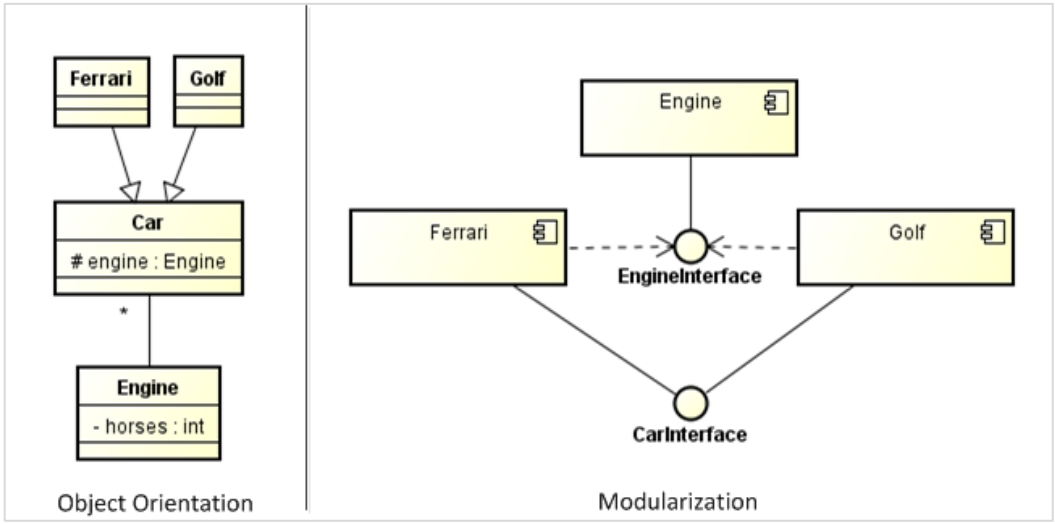



**Figura 8 – Diferença entre Modularização e Orientação a Objetos**

Para HALL *et al.* (2010), a modularização e orientação a objetos definem maneiras diferentes de implementação porém com os mesmos propósitos, conforme **Figura 9** a modularização e a orientação a objetos oferecem: visibilidade, acessibilidade, coesão e acoplamento.

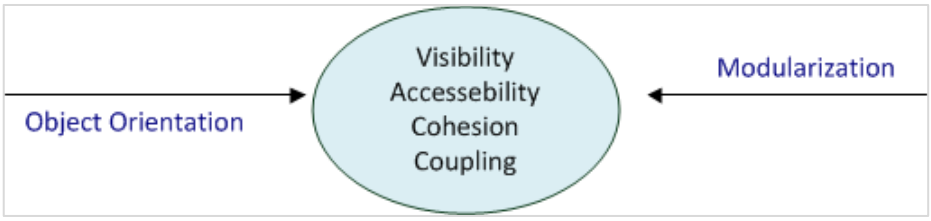



**Figura 9 – Propósitos da Orientação a Objetos e Modularização**

**Fonte: HALL *et al.*(2010)**

### 2.3.2 A limitação da Modularização em Java

Um dos problemas da linguagem Java é a falta de suporte nativo a modularização. Nessa seção serão abordados problemas no controle de visibilidade, desorganização do *Classpath* e problemas de implantação e manutenção. É apresentado como o *framework* OSGi pode resolve-los de uma maneira adequada e organizada.

#### 2.3.2.1 Controle de visibilidade

A linguagem Java provê alguns modificadores de visibilidade como: `public`, `private`, `protected` e restrição de pacote (sem modificador). Segundo HALL *et al.* (2010), o conceito de pacote em Java se restringe a dividir código em pastas separadas. E de fato é, para que uma classe possa ser visível a classes de outros pacotes ela deve possuir o modificador `public`, ou seja, a separação de classes por pacote serve apenas para organização física de arquivos.

Seguindo esse raciocínio, pelo fato de que uma classe precisa ser exposta publicamente para outras classes de um mesmo *jar*, faz com que ela seja exposta também para classes de outros *jars*. Conforme pode ser observado na **Figura 10**, existem dois arquivos *jars*: `bundleA.jar` e `bundleB.jar`. O *bundleB* possui uma interface pública (`ProvidedInterface`), cujo proposito é ser visível para outros *bundles*. `ClassA` é a classe de implementação dessa interface, ela é dependente da `ClassB`. Para que seja possível fazer essa associação, a `ClassB` deve possuir o modificador `public`, uma vez que elas estão em pacotes distintos.

Para que o *bundleA* possa acessar a interface pública de *bundleB*, deverá inseri-lo em seu *classpath*, tornando-se assim um único *bundle*. Conforme representado na **Figura 10**, a `ClassC` possui acesso direto as classes `ClassA` e `ClassB`. Para HALL *et al.* (2010), isso expõe detalhes de implementação que não deveriam ser expostos, sendo possível essas classes serem acessadas diretamente por classes externas, quando não poderia, podendo ocorrer problemas em futuras evoluções da aplicação.

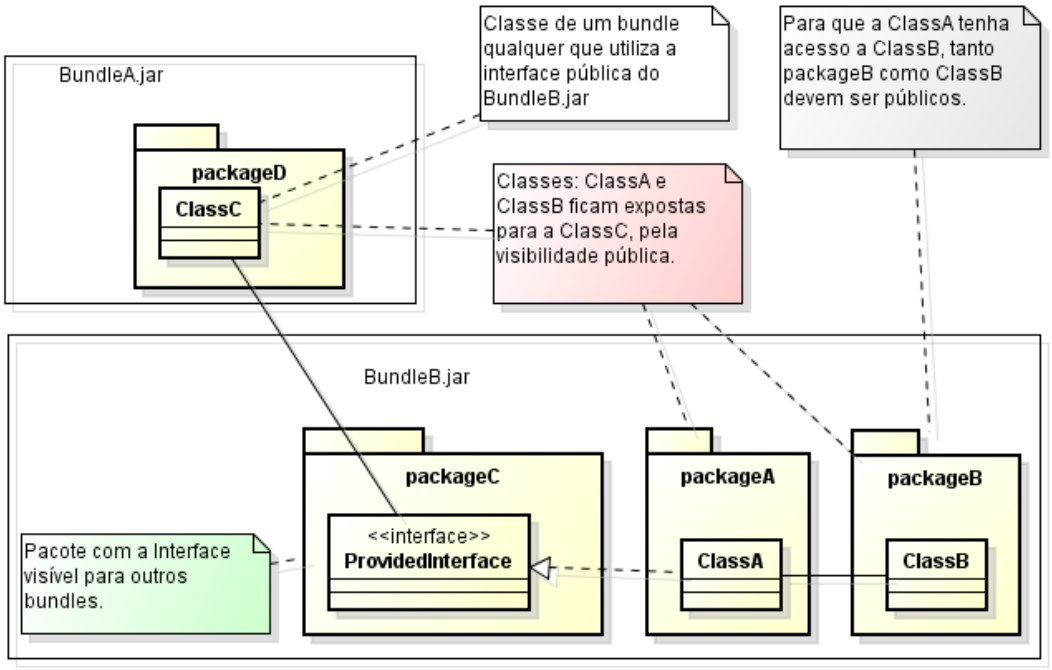



**Figura 10 – Visibilidade de classes para outros *bundles***

O OSGi propõe mais um nível de visibilidade, mantendo os anteriores da maneira como sempre foram utilizados, expondo publicamente as classes para que fiquem visíveis a outros pacotes. Esse novo nível funciona como um filtro, por padrão todos os pacotes serão ocultados para os demais *bundles*, expondo somente determinados pacotes. Para Hall *et al.* (2010), as implementações de um módulo devem ficar protegidas dos demais módulos, apenas as classes de acesso devem ser explicitamente expostas. Observa-se na **Figura 11** que há três classes protegidas (`ClassA`, `ClassB` e `ClassE`) e duas classes expostas (`ClassC` e `ClassD`).

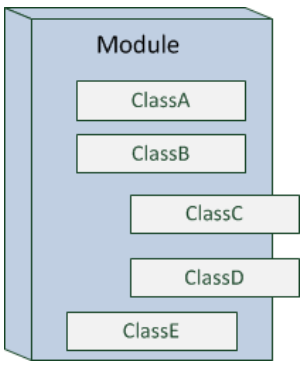



**Figura 11 – Esqueleto de um módulo**

**Fonte: HALL *et al.* (2010)**

Cada arquivo *jar* contém um arquivo chamado MANIFEST.MF que deve estar no diretório META-INF. Conforme ORACLE (2010), trata-se de um arquivo especial que contém informações sobre os arquivos empacotados. Nesse arquivo são inseridas informações que são interpretadas pelo *framework* OSGi, no caso dos pacotes exportados, eles são definidos pela propriedade `Export-Package`, onde os pacotes são separados por vírgula, todas as classes públicas desses pacotes são expostas (HALL *et al.*, 2010).

Aplicando esse conceito ao `BundleB.jar`, pode se observar na **Figura 12** que utilizando o *framework* e adicionando a propriedade no arquivo de manifesto, apenas o pacote `packageC` fica visível ao *BundleC*.

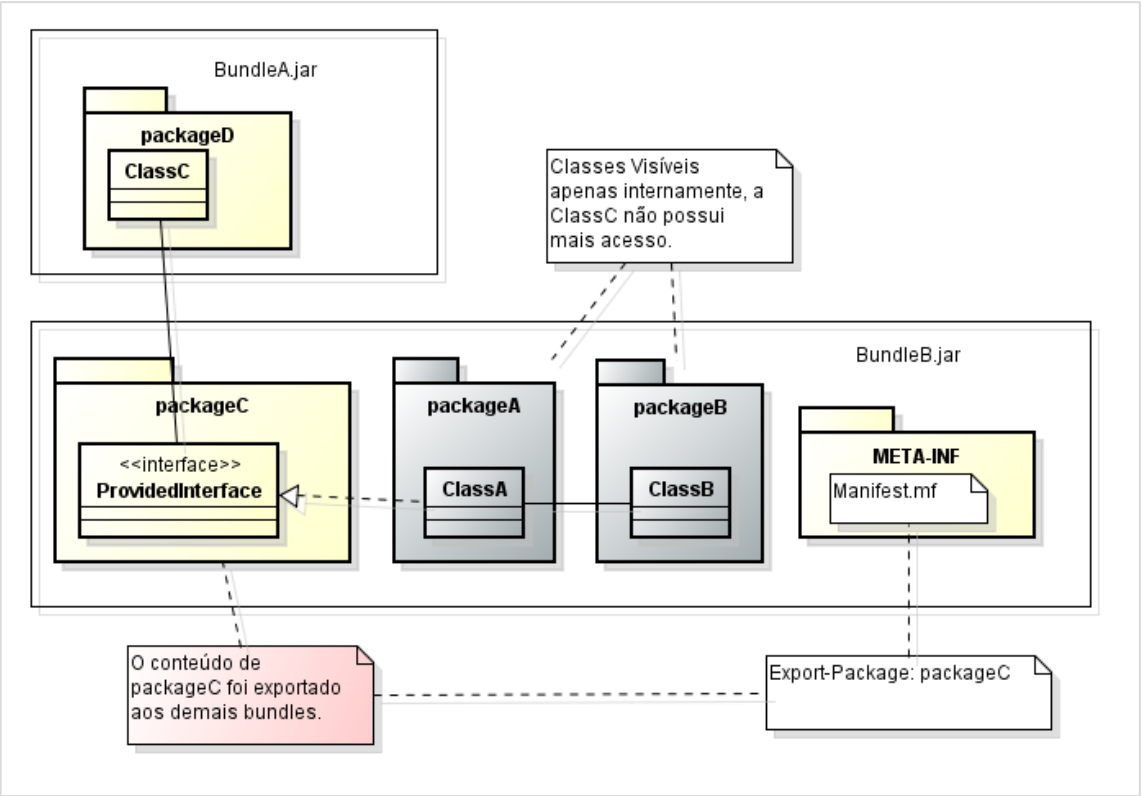



**Figura 12 – Alteração da visibilidade com o *framework* OSGi**

### 2.3.2.2 Desorganização do *classpath*

Outro grande problema da linguagem Java é a desorganização do *classpath*. Quando em determinada aplicação que possui várias bibliotecas dependentes, ao iniciar a aplicação, a máquina virtual faz o mapeamento de todas as classes para o *classpath*. Até aí tudo bem, o

problema é que se houver a mesma classe em dois ou mais *jars*, a máquina virtual utilizará a primeira classe que ele encontrar, as outras ele simplesmente descarta mesmo que sejam versões diferentes. Para Hall *et al*. (2010), o *classpath* não presta atenção em versionamento de classes, ele simplesmente retorna a primeira classe que ele encontrar.

Isso pode ser observado na **Figura 13**, onde há os *jars*: `Jar1`, `Jar2`, `Jar3`. Cada figura geométrica representa um determinado tipo de classe. Se a aplicação fosse executada adicionando a seguinte linha no arquivo de MANIFEST.MF: `Class-path: Jar1.jar Jar2.jar Jar3.jar`, o Hexágono utilizado será do `Jar1`, Circulo do `Jar1`, Triangulo do `Jar2` e Estrela do `Jar3`.

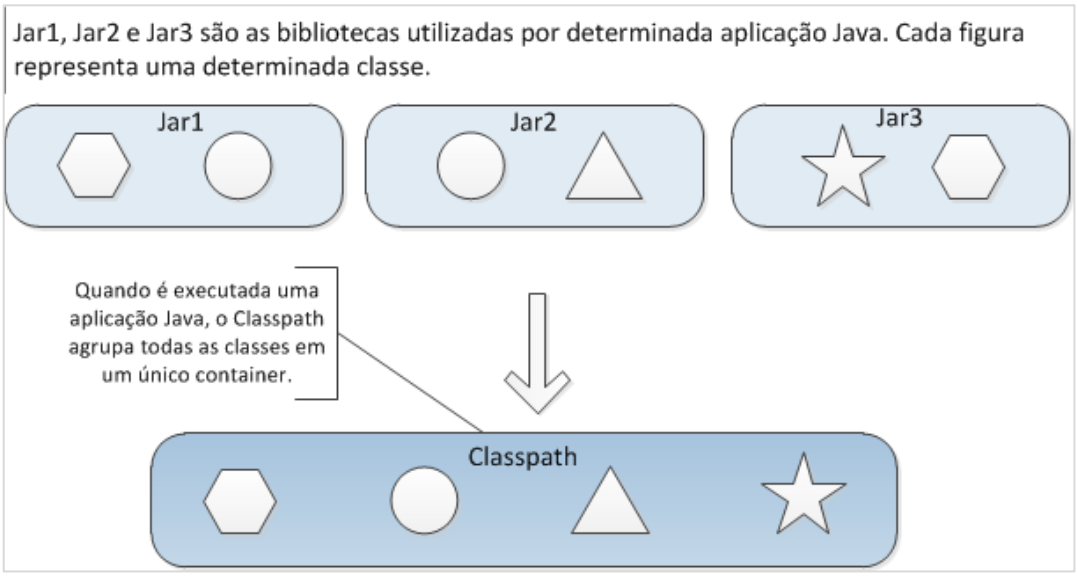



**Figura 13 – *Classpath* após inicialização da aplicação**

**Fonte: HALL *et al.* (2010)**

Para comprovar isso foi desenvolvido um projeto no eclipse que representa essa problemática, **Figura 14**.

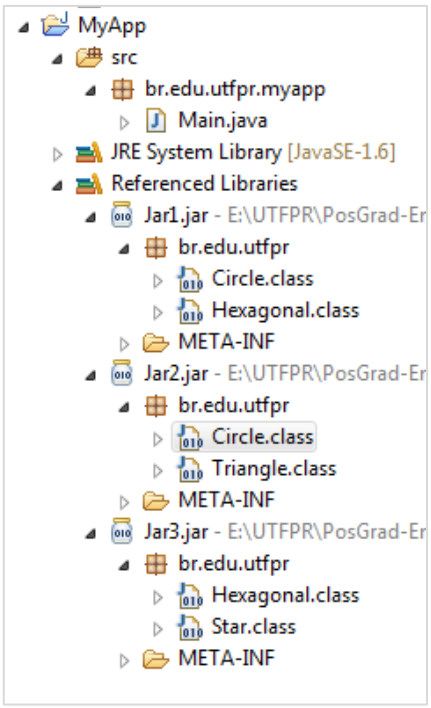



**Figura 14 – Projeto com o problema do *classpath***

A classe `Main` conforme **Figura 15**, possui referência à classe `Circle`. Conforme pode ser observado no editor de classe, está sendo utilizada a classe `Circle` do `Jar1`, que é primeira biblioteca referenciada no projeto. Isso é um grande problema, pois se for necessário utilizar a outra versão da classe `Circle` presente no `Jar2`. A ordem que é adicionada uma biblioteca adicional, **Figura 14**, implica na busca por determinada classe, ou seja, se inverter o `Jar1` e o `Jar3`, o resultado desse exemplo seria totalmente diferente.

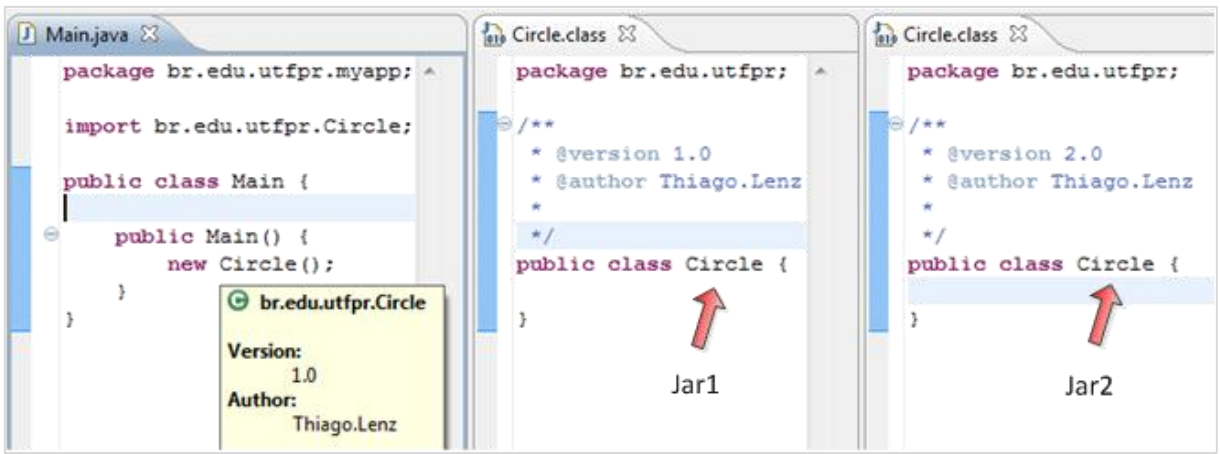



**Figura 15 – Origem da classe *Circle***

O OSGi implementa o mecanismo de versionamento de pacotes. Bem como existe a configuração da propriedade `Export-package` que determina quais pacotes serão expostos, existe a configuração da propriedade `Import-package` que importa pacotes de outros *bundles*. Nessas duas propriedades é possível determinar qual versão do pacote está sendo exportada e qual versão esta sendo importada. Conforme pode ser observado na **Figura 16**, o `Jar1` e o `Jar2` exportam o mesmo pacote: `br.edu.utfpr`, porém em versões diferentes. Já o *myapp.jar* apresenta como é a importação das duas versões desse pacote.

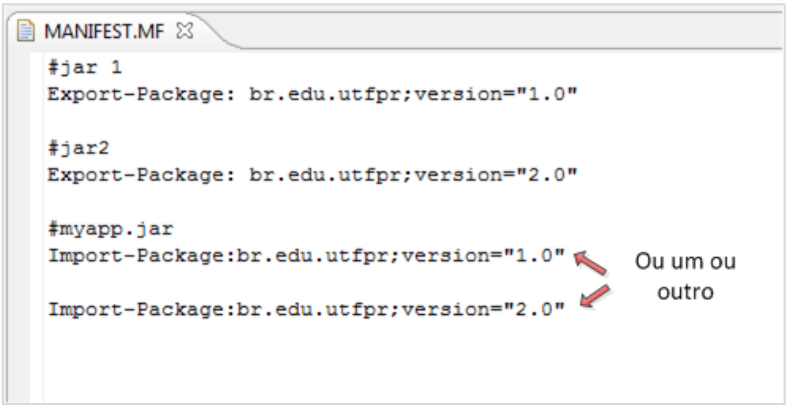



**Figura 16 – Exemplo de versionamento de pacotes**

O exemplo de como utilizar essa solução foi relativamente simples, no item **2.3.3.1.2 - Propriedades e configurações do arquivo Manifesto**, há mais detalhes sobre a questão do arquivo de manifesto, e no item **2.3.3.1.7 – Versionamento** é abordado sobre as versões dos *bundles*.

### 2.3.2.3 Dificuldade na implantação e manutenção de um aplicativo em Java

Em qualquer aplicativo Java de pequeno a grande porte possui inúmeras dependências de outras bibliotecas, como por exemplo: *Hibernate*, *Spring*, ou ainda, as diversas bibliotecas do *Apache Commons*. Essas bibliotecas possuem outras bibliotecas como dependentes. Em determinada aplicação Java, adicionando o *Hibernate* sem suas dependências no *classpath*, a aplicação compilará, mas na execução em algum ponto qualquer de alguma classe do

*Hibernate* que utilize uma dessas dependências não adicionadas, a JVM lançará uma exceção do tipo `ClassNotFoundException`.

A **Figura 17** representa a situação reportada no paragrafo anterior, onde se tem uma aplicação (*MyApp1.0*) que possui dependência com o *Hibernate3.5*, o qual possui dependência com outras bibliotecas. Se não forem adicionadas tais dependências no *classpath*, ocorrerá um erro de execução. Isso acontece pelo fato do compilador Java não avançar até as dependências de 2º nível ou mais da aplicação *MyApp1.0*, uma vez que as dependências de primeiro nível (*Hibernate3.5*) já foram compiladas.

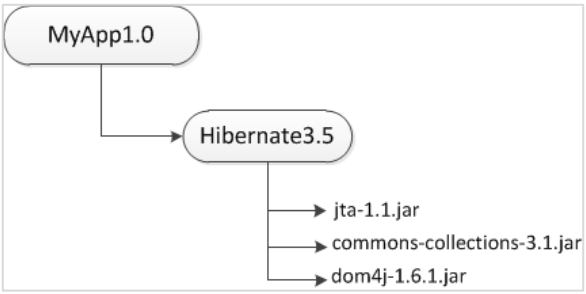



**Figura 17 – Exemplo de dependências**

Outro problema que ocorre também é na manutenção. Em todas as aplicações Java que estejam em produção, onde for necessário atualizar um determinado componente, deverá ser finalizada a execução da aplicação. Isso será necessário para qualquer atualização que seja feita, pois o *classpath* fará a leitura e mapeamento de classes novamente, mesmo que não seja adicionada nenhuma classe nova.

A **Figura 18** ilustra essa situação, a aplicação *MyApp-1.0* possui três módulos: *FinancialModule-1.0*, que possui as funcionalidades de finanças e controle de caixa; *HumanModule-1.0*, que disponibiliza funcionalidades de recursos humanos, folha de pagamento e controle de férias; e *PrinterModule-1.0*, que realiza o controle de fila de impressão. Conforme esse sistema é utilizado pelo cliente encontram-se alguns bugs no módulo de impressão. Tais *bugs* são corrigidos na versão 1.0.1 do *PrinterModule*. Para instalar essa atualização será necessário parar a aplicação, substituir o componente pelo novo e iniciar a aplicação. Essas atualizações se tornam um problema quando há um requisito de disponibilidade de serviço próximo a 100%.

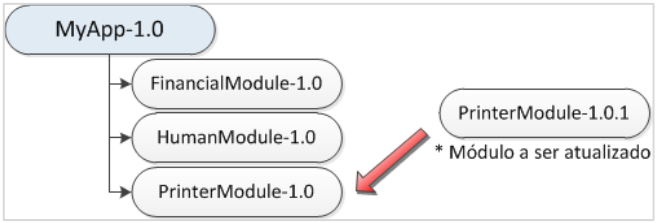



**Figura 18 – Atualização de um módulo da aplicação**

Esses dois problemas podem ser minimizados utilizando os níveis mais avançados do *framework* OSGi. Conforme THE OSGi Architecture (2011), utilizando o nível de ciclo de vida os componentes podem ser instalados, iniciados, atualizados, parados e desinstalados. Quanto ao problema das bibliotecas que não foram adicionadas ao projeto, o OSGi garante a consistência das dependências de todos os *bundles*. Conforme HALL *et al*. (2010), ao iniciar a aplicação o *framework* executa um algoritmo de verificação das dependências com efeito cascata, onde em caso de falha não será possível inicializar os componentes cujas dependências não foram encontradas.

Utilizando o ciclo de vida não é necessário parar toda a aplicação para atualizar determinado componente ou instalar uma nova versão. É possível realizar a atualização e em seguida apenas acionar um comando para os demais componentes se atualizarem perante a nova versão disponível. Segundo HALL *et al*. (2010), para minimizar o impacto da atualização ela é feita em duas etapas: a primeira etapa prepara o ambiente, e a segunda efetiva e recalcula as dependências entre os componentes.

Como pode ser observado na **Figura 19**, é iniciada uma aplicação (#1) que possui dois *bundles*, *BundleA* na revisão 1 e *BundleB* na revisão 1 também, onde *BundleB* possui dependência para o *BundleA*. Em determinado momento é instalada uma nova revisão do *BundleA*. Nessa etapa (#2) o *classpath* possui duas revisões do *BundleA*, e os *bundles* que posuírrem dependência para este continuaram referenciando a revisão 1. Na segunda parte do processo (#3) é realizada a efetivação da atualização através do comando *refresh*. Com esse comando todas as dependências são recalculadas, onde agora será utilizada a nova versão do componente que substituiu a antiga.

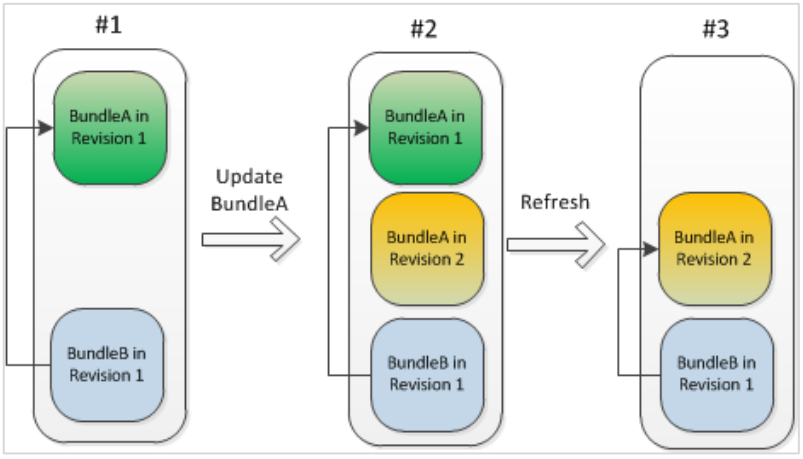



**Figura 19 – Atualização de *bundles* em tempo de execução**

**Fonte: HALL *et al.* (2010)**

### 2.3.3 Arquitetura

Nesse item será detalhando sobre os três níveis de utilização Modularização, Ciclo de vida e Serviços. Em cada nível serão apresentados os conceitos básicos de utilização. Os níveis são apresentados na sequência do mais básico ao mais avançado, uma vez que cada nível é dependente dos anteriores. A **Figura 20** representa as camadas do *framework*, a base de tudo é a camada de hardware e o sistema operacional, em seguida vem o núcleo (*Execution Environment*), camada de modularização, Ciclo de vida e Serviços. Paralelo a isso existe a camada de segurança, que é opcional.

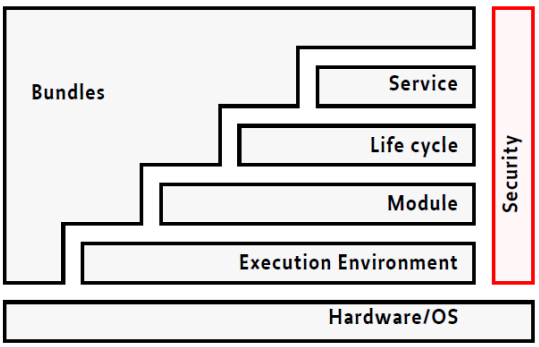



**Figura 20 – Arquitetura de camadas do OSGi**

**Fonte: OSGi Alliance Specifications (2011)**

#### 2.3.3.1 Modularização (*Modularization*)

Nesse item será abordado sobre a camada básica de utilização do OSGi. Primeiramente é explicado o conceito de *bundle*. Após isso é realizado um breve resumo sobre as configurações mais importantes no arquivo de manifesto. Na sequência explica-se como utilizar a classe de ativação, que será executada na inicialização do componente. Em seguida trabalha-se na questão das dependências entre os *bundles* e por fim é explicado o conceito de versionamento.

##### 2.3.3.1.1 O conceito de *bundle*

O *bundle* nada mais é que a unidade de manipulação do *framework* OSGi, pode ser entendido como componente, módulos e plug-ins. Para GÉDÉON (2010), é muito semelhante a um arquivo *jar* comum, onde a principal diferença são as informações adicionadas no arquivo de manifesto que são interpretadas pelo OSGi.

A **Figura 21** representa a estrutura de um OSGi *bundle*. Comparando-se com um arquivo *jar* normal, onde todas as classes e arquivos são expostos aos demais, nessa estrutura são definidos os pacotes expostos e os protegidos dos demais *bundles*. A vantagem de utilizar essa estrutura é o fato do *bundle* poder ser utilizado em qualquer aplicação, seja ela baseada no *framework* ou não.

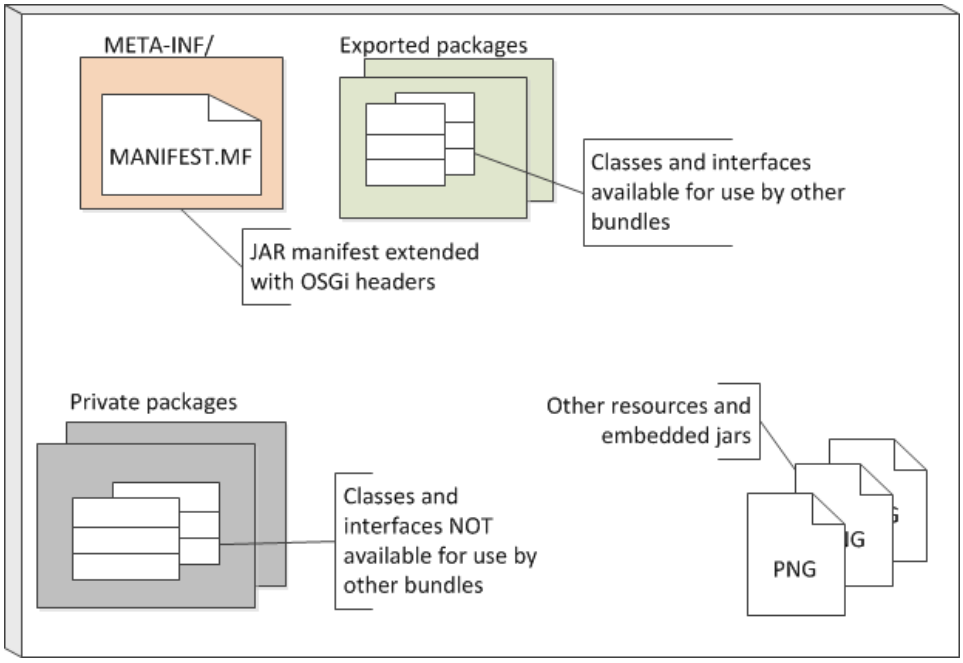



**Figura 21 – Estrutura de um *bundle***

**Fonte: GÉDÉON (2010)**

2.3.3.1.2 Propriedades e configurações no arquivo de MANIFESTO

O arquivo MANIFEST.MF nada mais é que um arquivo de propriedades (chave e valor), ele possui estrutura idêntica aos arquivos *.properties*. Nesse arquivo são inseridas configurações interpretadas pelo *framework* OSGi. Elas determinam o comportamento de cada *bundle*. Conforme GÉDÉON (2010) as propriedades podem ser classificadas como: obrigatórias, funcionalidades e informações.

A **Figura 22** apresenta a sintaxe desse arquivo. A linha 1 apresenta o conceito de `name` e `value`. Já na linha 3 apresenta uma prática adotada pelo OSGi: `Property-Name`, onde para cada propriedade pode ser passada uma ou mais cláusulas (`clause`) separadas por vírgula. Finalmente na linha 5 é exibido um conceito mais complexo, onde a cláusula de cada propriedade pode ser formada por valor e parâmetros, os quais são separados por ponto e vírgula. Os parâmetros possuem uma sintaxe diferente, o valor é atribuído com o sinal de igual.

```
name: value

Property-Name: clause, clause, clause

Property-Name: target1; parameter1=value1; parameter2=value2,
target2; parameter1=value1; parameter2=value2,
target3; parameter1=value1; parameter2=value2

```

**Figura 22 – Sintaxe do arquivo de Manifesto**

**Fonte: HALL *et al.* (2010)**

O **Quadro 1** apresenta um resumo sobre as principais propriedades configuráveis no arquivo de manifesto.

| **Propriedades do arquivo MANIFEST.MF** | | |
|---|---|---|
| **Campo** | **Descrição** | **Exemplo de valor** |
| Campos obrigatórios | | |
| Bundle-ManifestVersion | Determina a versão das regras que o arquivo está seguindo. Deverá ser informado "1" para versões 3 e ou "2" para versões 4 ou superiores. | 2 |
| Bundle-SymbolicName | Define o domínio do *bundle*, por convenção se utiliza o domínio revertido. | br.edu.utfpr.printermodule |
| Campos relativos a Funcionalidades | | |
| Bundle-Version | Versão do *bundle*. Caso não for informado o valor padrão é 0.0.0 | 1.1.20 |
| Export-Package | Lista de pacotes que são expostas aos demais *bundles*, os valores são informados separados por virgula. É possível informar a versão de cada pacote. | br.edu.utfpr.printermodule ;version=1.0.1, br.edu.utfpr.financemodule;version=1 |
| Import-Package | Define os pacotes que são importados. São dependências que devem ser atendidas para que o *bundle* possa ser utilizado. Possui mesma sintaxe do Export. | br.edu.utfpr.humanmodule ;version=1.0.0 |
| Bundle-Activator | Nome completo da classe que implementa a interface org.osgi.framework.BundleActivator. Essa classe será executada quando o *bundle*. | br.edu.utfpr.printermodule .PrinterActivator |
| Bundle-Classpath | Poderá ser adicionado algum jar interno ao *bundle*. Os valores são separados por virgula. Não é um campo obrigatório, mas caso seja informado estará sobrescrevendo o valor padrão que é o ponto (".."). Logo o mesmo deverá ser informado juntamente com s demais valores. | Bundle-Classpath: . , /lib/extra*bundle*.jar |
| **Campos de Informação** | | |
| Bundle-Name | Nome do Bundle | Printer Module |
| Bundle-Description | Tem a finalidade de detalhar o propósito do Bundle. | Este componente gerencia a fila de impressão. |

**Quadro 1 – Algumas propriedades configuráveis do OSGi**

**Fonte: OSGi Alliance Specifications (2011)**

Existem ainda outras propriedades, muitas delas são para o uso mais avançado do *framework* que não serão apresentadas nesse trabalho. A **Figura 23** demonstra um exemplo prático da configuração de um *bundle*.

```
1  Bundle-ManifestVersion: 2
2  Bundle-SymbolicName: org.foo.shape.impl
3  Bundle-Version: 2.0.0
4  Bundle-Name: Simple Shape Implementations
5  Import-Package: javax.swing, org.foo.shape; version="2.0.0"
6  Export-Package: org.foo.shape.impl; version="2.0.0"
```

**Figura 23 – Exemplo de um arquivo MANIFEST.MF**

**Fonte: HALL *et al.* (2010)**

2.3.3.1.3 Ativação do Bundle

No nível de modularização quando o *framework* é inicializado, os *bundles* devem ser instalados e inicializados. Quando o *framework* inicia o *bundle* é verificada a existência da classe configurada na propriedade `Bundle-Activator`. Utilizando-se da implementação *Apache Felix*, foi desenvolvido um exemplo de *bundle* com essa classe. As configurações do arquivo de manifesto podem ser visualizadas na **Figura 24**. Detalhe para a linha 6 onde é configurada a classe de ativação, e na linha 7 onde é informado a importação do pacote do *framework.*

```
1  Bundle-ManifestVersion: 2
2  Bundle-Name: First TCC App
3  Bundle-Version: 1.0.0
4  Bundle-SymbolicName: br.edu.utfpr.firstapp
5  Export-Package: br.edu.utfpr.msg;version="1.0"
6  Bundle-Activator: br.edu.utfpr.firstapp.FirstBundleActivator
7  Import-Package: org.osgi.framework
```

**Figura 24 – Arquivo de manifesto do *FirstBundle.jar***

A **Figura 25** apresenta as duas classes desenvolvidas para esse componente: A classe `FirstBundleActivator` que implementa a interface `BundleActivator`, e a classe utilitária `Messages`. Nas linhas 15 e 20 da classe de ativação são obtidos valores do contexto do *framework* através da classe `BundleContext`. Através dessa classe é possível obter a

referência do objeto *bundle* bem como dos demais. Em ambas as linhas será obtido o valor da propriedade `SymbolicName`.

FirstBundleActivator.java

```
1  package br.edu.utfpr.firstapp;
2
3  import org.osgi.framework.BundleActivator;
4  import org.osgi.framework.BundleContext;
5
6  import br.edu.utfpr.msg.Messages;
7
8  public class FirstBundleActivator implements BundleActivator{
9
10     private Messages msg;
11
12     @Override
13     public void start(BundleContext bc) throws Exception {
14         msg = new Messages();
15         msg.msgStart(bc.getBundle().getSymbolicName());
16     }
17
18     @Override
19     public void stop(BundleContext bc) throws Exception {
20         msg.msgEnd(bc.getBundle().getSymbolicName());
21     }
22 }
```

Messages.java

```
1  package br.edu.utfpr.msg;
2
3  public class Messages {
4      public void msgStart(String app){
5          System.out.println("Starting: "+ app);
6      }
7
8      public void msgEnd(String app){
9          System.out.println("Finishing: "+ app);
10     }
11 }
```

**Figura 25 – As duas classes do componente *FirstBundle***

Após o empacotamento em um arquivo *jar*, copia-se esse *jar* para a pasta `E:\Felix\felix-framework-4.0.1\bundle`. Em seguida é acessado o diretório (`E:\Felix\felix-framework-4.0.1`) através da linha de comando do sistema operacional, conforme **Figura 26**. Estando nesse diretório executa-se o *Apache Felix* através do comando da linha 1. O *Apache Felix* está configurado para instalar e inicializar todos os *bundles* presentes no diretório *bundle*. Ao iniciar o *bundle FirstBundle* é realizada uma chamada para o método `start()`, onde o resultado é apresentado na Linha 2.

Nesse momento todos os *bundles* estão ativos, como pode ser visto pelo resultado do comando `lb` (*list bundles*), linha 6 a linha 14. A finalização da aplicação se dá através do comando `stop 0` (ID do *bundle* principal OSGi). Nesse momento será realizada uma chamada de método para o método `stop()` de todos os *bundles* ativos e que possuírem configurada a classe de ativação. Para o *bundle* desse exemplo a saída pode ser visualizada na linha 17.

```
1  E:\Felix\felix-framework-4.0.1>java -jar bin/felix.jar
2  Starting: br.edu.utfpr.firstapp
3  ____________________________
4  Welcome to Apache Felix Gogo
5
6  g! lb
7  START LEVEL 1
8     ID|State      |Level|Name
9      0|Active     |    0|System Bundle (4.0.1)
10     1|Active     |    1|First TCC App (1.0.0)
11     2|Active     |    1|Apache Felix Bundle Repository (1.6.6)
12     3|Active     |    1|Apache Felix Gogo Command (0.12.0)
13     4|Active     |    1|Apache Felix Gogo Runtime (0.10.0)
14     5|Active     |    1|Apache Felix Gogo Shell (0.10.0)
15
16 g! stop 0
17 g! Finishing: br.edu.utfpr.firstapp
18
19 E:\Felix\felix-framework-4.0.1>
```

**Figura 26 – Texto da console da aplicação**

A presença dessa classe de ativação não é obrigatória, ela é muito útil quando o componente possui algumas tarefas a serem executadas no momento de sua inicialização, como por exemplo, instanciação de objetos, instalação e obtenção de referência de serviços, leitura de arquivos, etc.

#### 2.3.3.1.4 Dependências entre *Bundles*

A resolução se da ao momento que os *bundles* são instalados. Essa resolução acontece transitivamente e em efeito cascata. Quando essa resolução acontece cria-se uma ligação (*Wired*) entre os dois *bundles*, onde um importa determinado pacote exportado por outro. Para HALL *et al.* (2010), o resultado é um gráfico de componentes interligados através de fios de conexão, onde todas as dependências estão satisfeitas. Onde nenhum *bundle* poderá ser inicializado até que todas suas dependências obrigatórias sejam resolvidas.

Nesse item será abordado sobre o funcionamento da resolução de dependências simples onde há um único fornecedor de um determinado pacote. Em seguida é apresentado um exemplo mais complexo quando há dois *bundles* exportando o mesmo pacote. E por fim é apresentado como manipular a resolução através de restrições de versão.

Conforme visto no **Quadro 1**, a configuração das dependências é realizada através das propriedades `Import-Package` e `Export-Package`. Para uma simples dependência serão considerados dois *bundles*: *Calculator*, que exporta o pacote `br.edu.utfpr.calculator;version="1.0"`; *BundleA*, no qual importa o pacote `br.edu.utfpr.calculator`, **Figura 27**.

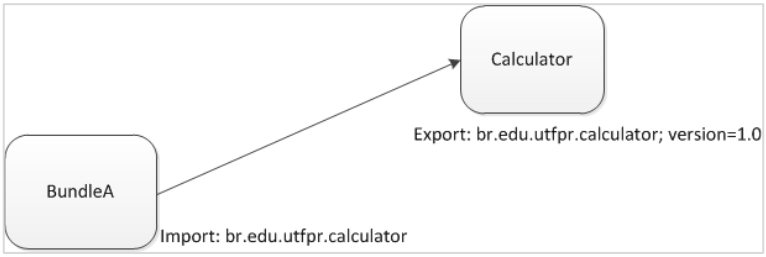



**Figura 27 – Simples dependência**

Para tal exemplo será criado um arquivo *jar* para cada *bundle*. Na **Figura 28**, está representado os arquivos de manifesto de cada *jar*. Destaca-se algumas configurações importantes como na linha 6 onde é informada a exportação de pacotes do *bundle Calculator* e na linha 14 onde é realizada a importação dos pacotes utilizados no *BundleA*.

```
1  #calculator Manifest
2  Bundle-ManifestVersion: 2
3  Bundle-Name: Calculator
4  Bundle-Version: 1.0.0
5  Bundle-SymbolicName: br.edu.utfpr.calculator
6  Export-Package: br.edu.utfpr.calculator;version="1.0"
7
8  #BundleA Manifest
9  Bundle-ManifestVersion: 2
10 Bundle-Name: Bundle A
11 Bundle-Version: 1.0.0
12 Bundle-SymbolicName: br.edu.utfpr.bundlea
13 Bundle-Activator: br.edu.utfpr.bundlea.ActivatorBundleA
14 Import-Package: org.osgi.framework, br.edu.utfpr.calculator
```

**Figura 28 – Manifesto dos *bundles BundleA* e *Calculator***

Após a configuração dos dois arquivos realiza-se a implementação de duas classes: `ActivatorBundleA` e `Calculator`. A classe `Calculator` situada no pacote `br.edu.utfpr.calculator` possui um método `sum(float, float)` que realiza um processamento de soma evidentemente. Já a classe `ActivatorBundleA` situada no pacote `br.edu.utfpr.bundlea` implementa a interface `BundleActivator`. Na linha 14 dentro do método `start()` é instanciado a variável `Calculator`, **Figura 29**.

```
1  package br.edu.utfpr.bundlea;
2
3  import org.osgi.framework.BundleActivator;
4  import org.osgi.framework.BundleContext;
5
6  import br.edu.utfpr.calculator.Calculator;
7
8  public class ActivatorBundleA implements BundleActivator{
9      private Calculator calc;
10
11     @Override
12     public void start(BundleContext bc) throws Exception {
13         System.out.println("Starting the BundleA!");
14         calc = new Calculator();
15         System.out.println("Result of 1 + 3 = "+ calc.sum(1, 3));
16     }
17
18     @Override
19     public void stop(BundleContext bc) throws Exception {
20         System.out.println("Stoping the BundleA!");
21     }
22 }
```

```
1  package br.edu.utfpr.calculator;
2
3  public class Calculator {
4      public float sum(float a , float b){
5          return a + b;
6      }
7  }
```

**Figura 29 – Classe *ActivantorBundleA* e *Calculator***

A execução desse exemplo pode ser visualizada na **Figura 30**. Na linha 3 é apresentado o retorno da execução do método `sum()`. Já nas linhas 11 e 12 podem ser visualizados os dois *bundles* instalados e ativados com sucesso.

```
E:\Felix\felix-framework-4.0.1>java -jar bin/felix.jar
Starting the BundleA!
Result of 1 + 3 = 4.0
____________________________
Welcome to Apache Felix Gogo

g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (4.0.1)
    1|Active     |    1|Bundle A (1.0.0)
    2|Active     |    1|Calculator (1.0.0)
    3|Active     |    1|Apache Felix Bundle Repository (1.6.6)
    4|Active     |    1|Apache Felix Gogo Command (0.12.0)
    5|Active     |    1|Apache Felix Gogo Runtime (0.10.0)
    6|Active     |    1|Apache Felix Gogo Shell (0.10.0)

g! stop 0
g! Stoping the BundleA!
```

**Figura 30 – Saída da console para o exemplo de dependência simples**

A resolução de um pacote por padrão é obrigatória. Caso não seja, no arquivo de manifesto na propriedade de importação do pacote deverá ser adicionado o parâmetro `resolution` com o valor `optional`, exemplo: `Import-Package: p; resolution:=optional; version="1.3.0"`. Assim, caso não seja possível resolver o pacote, não falhará a inicialização do componente que está importando um pacote inexistente.

### 2.3.3.1.5 Dependência com mais de um fornecedor

Na dependência simples quando existe apenas um fornecedor para o pacote, o *framework* não encontrou dificuldades para resolver. Porém quando existem 2 ou mais fornecedores para determinado pacote conforme **Figura 31**, qual *bundle* será escolhido?

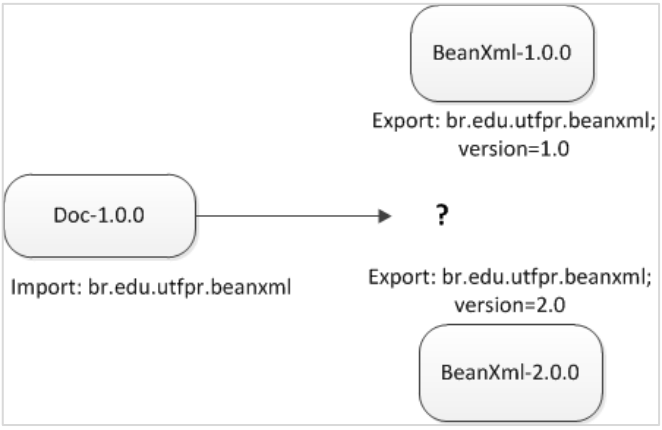



**Figura 31 – Qual *bundle* será utilizado?**

Conforme HALL *et al*. (2010), o *framework* dá prioridade aos exportadores já resolvidos, por isso, se ele tiver que escolher entre dois candidatos, onde um é resolvido e outro não, ele escolherá o resolvido. Essa definição pode ser visualizada nas regras abaixo:

1. A prioridade é para os *bundles* que já estejam instalados e resolvidos, ou seja, eles já estão sendo referenciados e utilizados;
2. Caso a primeira regra não for satisfeita, será comparada a maior versão. Exemplo: versão 2.2.0 é maior que versão 2.1.0;
3. E em caso de igualdade será escolhida por ordem de instalação, ou seja, o que possuir menor ID.

Para representar isso foram desenvolvidos os três *bundles* representados na **Figura 31**. Foi desenvolvido um exemplo bem simples, onde a única diferença é a existência de duas versões do mesmo *bundle*. A **Figura 32** exibe as configurações resumidas de cada *bundle*. Na linha 4 é configurada a importação de pacotes para o componente *Doc*, com destaque para o pacote `br.edu.utfpr.beanxml`. A exportação desse pacote é configurada nas linhas 10 e 16, sendo uma na versão 1.0 e 2.0 respectivamente.

```
#DOC
Bundle-SymbolicName: br.edu.utfpr.doc
Bundle-Activator: br.edu.utfpr.doc.DocBundleActivator
Import-Package: org.osgi.framework, br.edu.utfpr.beanxml

#BeanXML 1.0
Bundle-Name: Bean XML
Bundle-Version: 1.0.0
Bundle-SymbolicName: br.edu.utfpr.bundlexml
Export-Package: br.edu.utfpr.beanxml;version="1.0"

#BeanXML 2.0
Bundle-Name: Bean XML
Bundle-Version: 2.0.0
Bundle-SymbolicName: br.edu.utfpr.bundlexml
Export-Package: br.edu.utfpr.beanxml;version="2.0"
```

**Figura 32 – Configuração com *bundles* de versões diferentes**

Na parte Java foi desenvolvida a classe `XmlConverter` em duas versões. Ambas são idênticas, mudando apenas o conteúdo de cada método `String getVersion()`. A classe `DocBundleActivator` é uma classe que cria uma instância da classe `XmlConverter`, onde é chamado o método `getVersion()` desse objeto dentro do método `start()`, **Figura 33**.

**Versão 1 - XmlConverter**

```
package br.edu.utfpr.beanxml;

public class XmlConverter {
    public String getVersion(){
        return "Version of XmlConvert is: 1";
    }
}
```

**Versão 2 - XmlConverter**

```
package br.edu.utfpr.beanxml;

public class XmlConverter {
    public String getVersion(){
        return "Version of XmlConvert is: 2";
    }
}
```

**Método start de DocBeanActivator**

```
    public void start(BundleContext bc) throws Exception {
        System.out.println("Starting Doc...");
        System.out.println(new XmlConverter().getVersion());
    }
```

**Figura 33 – Classes *XmlConverter* e método *start* da classe *DocBeanActivator***

Colocando os três *bundles* no diretório `E:\Felix\felix-framework-4.0.1\bundle`, é iniciado o *Apache Felix* conforme o comando da linha 1 da **Figura 34**. Quando ocorrer a saída da console conforme linha 2, os *bundles* já foram resolvidos e inicializados. A linha 3 (em destaque) apresenta a saída da linha 13 do código da **Figura 33**, resolvido com a versão 2 do pacote `br.edu.utfpr.beanxml`. Ainda na **Figura 34** é executado o comando para listagem dos *bundles* instalados (linha 7), conforme saída da console, os três componentes desenvolvidos nesse exemplo estão devidamente instalados e ativos.

```
1  E:\Felix\felix-framework-4.0.1>java -jar bin/felix.jar
2  Starting Doc...
3  Version of XmlConvert is: 2
4
5  ____________________________
6  Welcome to Apache Felix Gogo
7
8  g! lb
9  START LEVEL 1
10    ID|State      |Level|Name
11     0|Active     |    0|System Bundle (4.0.1)
12     1|Active     |    1|Bean XML (1.0.0)
13     2|Active     |    1|Bean XML (2.0.0)
14     3|Active     |    1|Doc (1.0.0)
15   .....
16 g!
```

**Figura 34 – Saída da console com dois pacotes iguais**

Esse comportamento depende muito da ordem em que os componentes são instalados. Além da situação de como o *framework* vai decidir o fornecedor de um pacote, quando já houver um resolvido, existe a questão do cache. Para cada *bundle* no momento de sua instalação é criada uma pasta de cache. Ao parar e reiniciar o Felix primeiramente serão carregados os *bundles* em cache e apenas em seguida os novos, caso existam. Logo, caso seja adicionado uma nova versão de um componente não significa que o *framework* utilizará a nova versão. Existem duas possibilidades para contornar essa situação: A primeira é excluir todo o cache; a segunda é utilizar o próximo nível da arquitetura que é o ciclo de vida, onde os componentes são instalados, atualizados e desinstalados em tempo de execução[2].

[2] 2.3.3.2 – Ciclo de Vida (*Lifecycle*)

#### 2.3.3.1.6 Restrições de versões

Visto as regras de resolução quando há mais de um fornecedor para um pacote, existe um mecanismo para manipular esse comportamento, a restrição de versão. Conforme OSGi ALLIANCE SPECIFICATIONS (2011), o mecanismo de restrição de versão define na importação de determinado pacote um intervalo compatível ou uma versão inicial. No arquivo de manifesto, as restrições são configuradas o parâmetro `version` da propriedade `Import-Package`, muito semelhante ao mecanismo de exportação. A **Tabela 1** apresenta na coluna Valor com algumas configurações possíveis, e na coluna Predicado a respectiva lógica de restrição.

**Tabela 1 – Exemplos de intervalos de restrição de versões**

| Valor | Predicado |
|---|---|
| [1.2.3, 4.5.6) | 1.2.3 <= x < 4.5.6 |
| [1.2.3, 4.5.6] | 1.2.3 <= x <= 4.5.6 |
| (1.2.3, 4.5.6) | 1.2.3 < x > 4.5.6 |
| (1.3.4, 4.5.6] | 1.2.3 < x >= 4.5.6 |
| 1.2.3 | 1.2.3 <= x >= ∞ |

**Fonte: OSGi ALLIANCE SPECIFICATIONS (2011).**

A lógica de restrição funciona da seguinte maneira: Quando for informada uma versão inicial e final, significa que só será satisfeita a importação se um pacote exportado de versão "x" estiver nesse intervalo. Caso seja informada apenas uma versão significa que todas as versões a partir dessa são compatíveis. Caso não seja informada nenhuma versão o *framework* considera a partir de "0.0.0" ao infinito. Essas configurações de restrição estão acima da resolução através de *bundles* já resolvidos, caso a versão já resolvida não satisfaça essa restrição, será utilizado outro fornecedor.

No exemplo dos dois fornecedores do item anterior, poderia ser utilizada a seguinte sintaxe: `Import-Package: br.edu.utfpr.beanxml; version="[1.0.0, 2.0.0)"`. A utilização dos colchetes "[ ]" significa menor e igual "<=" ou maior e igual ">=". A utilização dos parênteses "( )" significa apenas menor "<" ou maior ">". Essa regra pode ser

visualizada na **Figura 35**, onde o *BundleA* não poderá referenciar o *BundleC* por causa da restrição de versão.

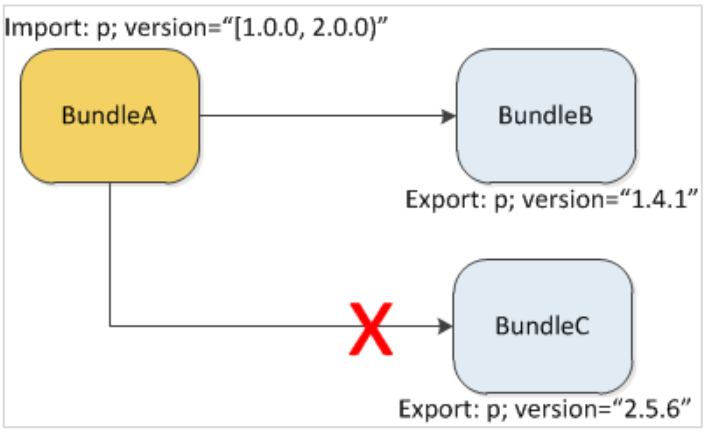



**Figura 35 – Restrição de importação de pacote**

**Fonte: OSGi ALLIANCE SPECIFICATIONS (2011)**

2.3.3.1.7 Versionamento

Esse item tem o propósito de explicar sobre o comportamento de versões no *framework* OSGi. Conforme HALL *et al*. (2010), uma versão é composta por 3 números separados por pontos: o primeiro representa a principal versão, a segunda representa mudanças médias no sistema, e a terceira serve para controle de correções de bugs. A principal versão quase não muda, geralmente é incrementada quando há uma mudança significativa no software/componente. A segunda é incrementada de acordo com as melhorias e adequações realizadas. Já a terceira é incrementada conforme as correções de bugs são liberadas.

É possível ainda adicionar ao final do terceiro número, adicionar uma palavra denominada *qualificador*. Nesse qualificador pode ser informado qualquer texto, conforme for necessário. A **Figura 36** apresenta como funciona o mecanismo de versionamento em ordem crescente.

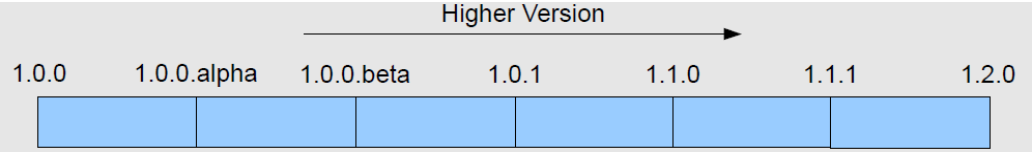



**Figura 36 – Mecanismo de versionamento**

**Fonte: HALL *et al*. (2010)**

Para o OSGi quando não for informada a versão no arquivo de manifesto, ele assume que a versão é a 0.0.0. Caso seja informada versão 1, ele considera o número 1.0.0, e assim por diante, é completado com zeros onde não for informado nenhum valor.

### 2.3.3.2 Ciclo de Vida (*Lifecycle*)

O item anterior apresentou uma abordagem do básico necessário para realizar o desenvolvimento de uma aplicação baseada em OSGi. Esse item aborda sobre o nível gerenciável, onde o mesmo é dependente do nível de modularização. Para HALL *et al.* (2010), esse nível permite realizar ações externas como: instalar, iniciar, atualizar, parar e desinstalar diferentes *bundles* da aplicação em tempo de execução.

Existem basicamente quatro fases de vida de um *bundle*: *Installation*, fase inicial onde o *bundle* é instalado no sistema. É nesse momento que são resolvidas suas dependências; *Execution*, caso a instalação seja bem sucedida o *bundle* pode ser executado ou utilizado por outros; *Update*, caso seja necessário atualizar o pacote com correções, é nessa fase que isso ocorre; *Removal*, caso o *bundle* não esteja sendo usado é possível desinstalá-lo, que caracteriza o fim do ciclo de vida, **Figura 37**.

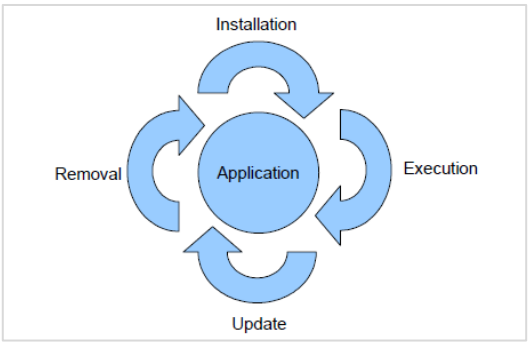



**Figura 37 – Fases de um *bundle***

**Fonte: HALL *et al* (2010)**

#### 2.3.3.2.1 Estados de um *Bundle*

Ao ser instalado no *framework* OSGi o *bundle* assume o estado INSTALLED. Em seguida é iniciado o processo de resolução de dependências obrigatórias o mesmo assume o estado RESOLVED, estando pronto para ser inicializado, caso contrário permanecerá no estado inicial até que as mesmas sejam corretamente configuradas. Ao ser inicializado através do comando *start*, passará para o estado STARTING. Nesse momento caso o *bundle* possua a classe de ativação configurada, será criada uma instância da mesma e chamado o método *start()*. Ao término de execução desse método o *bundle* estará estabilizado no estado ACTIVE.

Até aqui foi explicada a sequência de ida, agora será abordado sobre o retorno. Uma vez ativo o *bundle* poderá ser parado através do comando *stop*, passando para o estado STOPPING. Nesse momento será chamado o método *stop()* da instância criada da classe de ativação no momento da inicialização. Ao seu término o mesmo passará para o estado de RESOLVED.

Quando o *bundle* está no estado RESOLVED vários caminhos podem ser seguidos, ele poderá voltar para o estado INSTALLED, caso seja executado o comando *update* ou *refresh*, e ainda poderá ser desinstalado através do comando *uninstall*, passando para o estado de UNINSTALL, onde automaticamente será excluído, concretizando-se assim o fim do ciclo. No estado INSTALLED antes de ser resolvido, ou então, caso a resolução seja mal sucedida, é possível executar as ações de *refresh* e *update*, ou ainda o comando *uninstall*, sendo o bundle removido sem ser utilizado.

A **Figura 38** representa as transições de estados descritas até então. O item **2.3.3.2.3 – comandos** apresenta como realizar tais transições através de comandos definidos pelo *framework.*

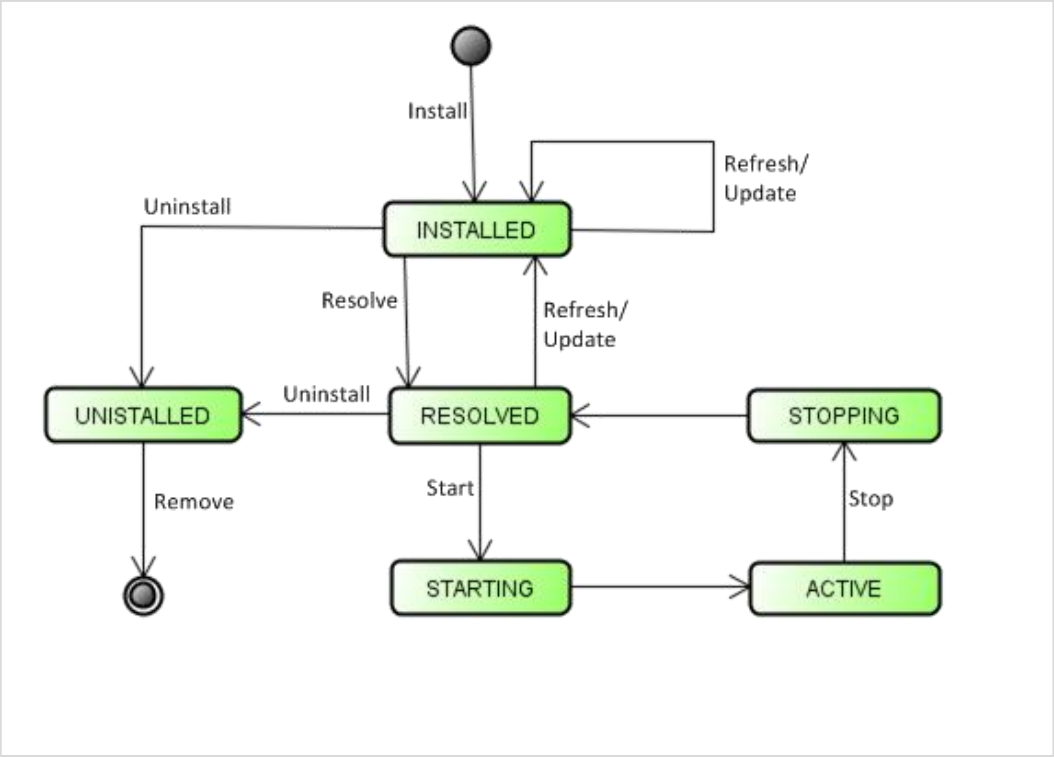



**Figura 38 – Diagrama de transição de estados de um *bundle***

**Fonte: OSGi ALLIANCE SPECIFICATIONS (2011)**

2.3.3.2.2 Um pouco sobre a API: Interfaces *BundleContext* e *Bundle*

Esse item abordará um pouco sobre as principais funções de duas interfaces da API de desenvolvimento: `BundleContext` e `Bundle`. Conforme OSGi ALLIANCE SPECIFICATIONS (2011), existe um objeto `BundleContext` para cada *bundle*, ele contém dados do contexto de execução, atua como proxy para acessar os demais *bundles* instalados. Essa instância é criada no momento de inicialização do *bundle* e disponibilizada através dos métodos da interface `BundleActivator`.

Suas principais funcionalidades são:

- Instalar novos *bundles*, através do método `installBundle`;
- Obter a referência do próprio objeto `Bundle`, bem como dos demais *bundles*;
- Publicar serviços bem como a obter a referência dos mesmos;
- Adicionar e disparar eventos.

O objeto `Bundle` por sua vez possui o papel de encapsular informações referentes ao *bundle*. As principais funcionalidades estão listadas abaixo:

- Atualizar;
- Parar;
- Desinstalar;
- Obter de referências de serviços utilizados;
- Obter atributos como: `id`, `state`, `symbolicName` e `version`;

Tendo conhecimento nessas duas interfaces já é possível realizar tranquilamente o desenvolvimento utilizando as camadas de modularização e ciclo de vida. Existem diversas outras interfaces que não foram abordadas nesse item, porém conforme necessário elas serão detalhadas ao decorrer do trabalho.

#### 2.3.3.2.3 Comandos

Essa seção tem por objetivo apresentar um dos propósitos do uso do ciclo de vida, o gerenciamento de componentes. Para tal o *framework* define algumas ações, as quais realizam a transação de um estado para outro, conforme **Figura 38**. Todos os comandos podem ser executados via linha de comando ou através de métodos das classes da API (*Application Programming Interface*).

O ponto de partida é a instalação que é realizada através do comando ***install***. Através da API, o *bundle* é instalado pelo método `installBundle(path)` do objeto `BundleContext`. O parâmetro desse método é o caminho absoluto do arquivo, exemplo: *e:\bundles\myBundle.jar*. Esse caminho pode ser também uma URL (*Uniform Resource Locator)*, por exemplo: `http://meusite.com.br/files/myBundle.jar`. É pós-condição desse comando que o *bundle* possua ou o estado de INSTALLED ou RESOLVED, caso contrário uma exceção do tipo `BundleException` será lançada informando que o mesmo não foi instalado.

Na linha de comando a instalação poderá ser feita através do comando do **Quadro 2**. Após a instalação, o *framework* atribui ao objeto `Bundle` as configurações contidas no arquivo manifesto e uma identificação (`id`) única.

```
Install file: e:/bundles/Beanxml-1.0.0.jar
install http://meusite.com.br/files/myBundle.jar
```

**Quadro 2 – Comando de Instalação via console**

Com o componente devidamente instalado é possível iniciá-lo através do comando ***start***. Utilizando-se da API, é possível iniciar um *bundle* através do método `start()` do objeto `Bundle`. Já no console é possível iniciá-lo através dos comandos descritos no **Quadro 3**. São duas possibilidades, ou é passado o id do *bundle* já instalado, ou então, existe a possibilidade de passar o caminho do arquivo, e caso o mesmo não esteja instalado, será feito nesse momento, trata-se de um diferencial que é permitido apenas na linha de comando. Utilizando a API, o *bundle* deverá ser instalado, obtido o objeto de referência (`Bundle`), e em seguida inicializado.

```
#(URL ou PATH)
start file:e:/bundles/Beanxml-1.0.0.jar
#ID do bundle
start 1
```

**Quadro 3 – Comandos de Console para iniciar um *bundle***

Existem algumas validações que são feitas ao ser executado o comando ***start***: caso as dependências obrigatórias não foram atendidas é lançado uma exceção do tipo `BundleException`; caso o *bundle* esteja desinstalado (UNINSTALLED) ou o mesmo está tentando iniciar a ele mesmo, será lançada uma exceção do tipo `IllegalStateException`. Caso todas as validações estejam satisfeitas, o *bundle* passará para o estado STARTING, onde será executado o método `start()` do objeto `BundleActivator`, e ao seu término passará para o estado ACTIVE (HALL *et al.* ,2010).

O comando ***stop*** realiza o processo inverso, quando um *bundle* estiver ativo o mesmo será parado. Primeiramente ele passará para o estado STOPPING, onde será executado o método `stop()` do objeto `BundleActivator`, e ao final voltará para o estado RESOLVED. Apenas uma validação é realizada. Caso o estado seja UNINSTALLED ou próprio *bundle* esteja tentando parar a si mesmo, será lançada uma exceção do tipo `IllegalStateException`.

Através da API o comando ***stop*** é semelhante ao comando ***start,*** onde é realizada uma chamada de método ao método `stop()` do objeto `Bundle`. Já na linha de comando do console é possível realiza-lo através do comando do **Quadro 4**.

```
stop 5
```

**Quadro 4 – Comando de console para parar um *bundle***

O comando para atualizar um *bundle* é o ***update***. Quando esse comando for executado primeiramente é verificado o estado do *bundle*, caso ele esteja ativo (ACTIVE), o *framework* internamente executará o comando ***stop***, devendo estar no estado RESOLVED ou INSTALLED para efetuar a atualização. Conforme HALL *et al.* (2010, esse comando resulta em uma nova revisão do *bundle*, sendo novamente atribuído a ele o estado de INSTALLED.

Pela API é a atualização é realizada através do método `update(InputStream)` do objeto `Bundle`, onde é passado um objeto do tipo `java.io.InputStream`. Pelo comando de console essa atualização é realizada através do comando `update`, semelhante ao comando `install`, **Quadro 5**.

```
#pelo java
Bundle b = bc.getBundle(1);
b.update(new FileInputStream(new File("e:/bundles/beanxml-1.0.0.jar")));
#pelo console é passado ID e path ou URL do arquivo
update 5 file:e:/bundles/beanxml-1.0.0.jar
```

**Quadro 5 – Comandos para atualizar um *bundle***

E por último, mas não menos importante o comando de desinstalação, ***uninstall***. A única validação que será feita nesse comando é a verificação se o *bundle* já não está desinstalado, caso estiver será lançada uma exceção do tipo `IllegalStateException`. Caso o *bundle* esteja ativo, será executado primeiramente o comando ***stop*** e somente então será desinstalado. Pela API a desinstalação pode ser feita através do método `uninstall()` do

objeto `Bundle`. Já pela console o mesmo pode ser feito através do comando `uninstall` do **Quadro 6**.

```
uninstall 5
```

**Quadro 6 – Comando de desinstalação**

### 2.3.3.3 Serviços (*Services*)

Por fim, a última camada de utilização do OSGi Framework, a camada de serviços. Essa camada trabalha em conjunto com as duas camadas apresentadas até então: modularização e ciclo de vida. Conforme OSGi ALLIANCE SPECIFICATIONS (2011), um serviço é um objeto Java normal que está registrado sob uma ou mais interfaces Java. Os *bundles* podem registrar serviços, buscá-los ou receber notificações quando um serviço mudar de status.

Com esse modelo de interface e implementação define-se um modelo de contrato entre o fornecedor e o consumidor. Conforme HALL *et al.* (2010) os consumidores não estão preocupados com a implementação exata por trás de um serviço, uma vez que os mesmos são de certa forma substituíveis, **Figura 39**.

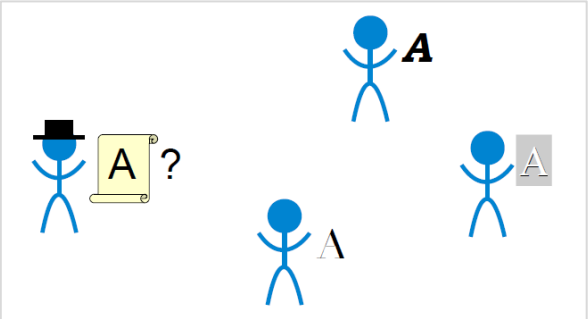



**Figura 39 – Contrato de interface e implementação**

**Fonte: HALL *et al.* (2010)**

A ligação entre componentes e serviços é de certa forma que um complementa o outro. Na camada de serviços um *bundle* pode conter um conjunto de serviços, estes devem ser registrados no registro de serviços do *framework* durante a inicialização do bundle, e desregistrados no momento da paralização. Ao registrar esses serviços, os mesmos ficam disponíveis para que outros *bundles* possam acessá-los, sendo que os acessos a esses, são gerenciados pelo *framework*, (OSGi ALLIANCE SPECIFICATIONS, 2011).

#### 2.3.3.3.1 As quatro grandes vantagens

Conforme HALL *et al*. (2010), existem 4 grandes vantagens adquiridas utilizando a camada de serviços: Baixo acoplamento, maior ênfase em interfaces, descrições extras dos serviços e suporte em tempo real de várias implementações.

O **baixo acoplamento** entre consumidores e provedores é uma das mais importantes dentre as quatro. O contrato informa somente as informações necessárias que devem ser conhecidas pelos consumidores. Esse modelo permite que seja possível haver mais de uma implementação desse contrato, facilitando assim a troca entre um serviço e outro. Um contrato define claramente o limite entre grandes componentes, e contribui para o desenvolvimento e manutenção dos mesmos.

A **maior ênfase em interfaces**, as interfaces oferecem um mecanismo melhor que a herança, uma vez que a classe concreta pode implementar várias interfaces porém não pode estender mais de uma classe. Elas também definem as assinaturas de métodos que um serviço deve fornecer. Uma vez definida tais interfaces, pode se iniciar o desenvolvimento dos consumidores sem precisar esperar até que os serviços estejam prontos, e vice-versa.

A terceira vantagem é permitir informar **descrições extras dos serviços**. Somente a interface às vezes não é suficiente para saber o que determinado serviço fará no outro lado. Para isso existe o conceito de dicionário de dados, ou meta-dados. Essas informações são passadas no momento de registro do serviço e podem ser utilizadas para filtrá-los no momento de descoberta dos mesmos.

Quanto à quarta vantagem, **suporte em tempo real de várias implementações**, consiste basicamente em haver várias implementações de um serviço. Em determinado momento, um desses serviços é retirado para atualização ou removido, automaticamente um

dos outros serviços passará a ser utilizado, sem a necessidade de parar o componente consumidor para resolver a nova dependência.

#### 2.3.3.3.2 Registro e descoberta de Serviços

Os serviços devem ser registrados durante a inicialização do *bundle*, através do método `start()` da classe que implementa a interface `BundleActivator`. O registro consiste basicamente em informar a(s) interface(s) que um determinado objeto está assinando. Esse registro é feito através dos métodos do objeto `BundleContext`, conforme linha 30 da **Figura 40**. Os parâmetros são: a(s) classe(s) da(s) interface(s); a instância do objeto que implementa essa(s) interface(s), e um objeto `Dictionary`, que não necessário para esse exemplo.

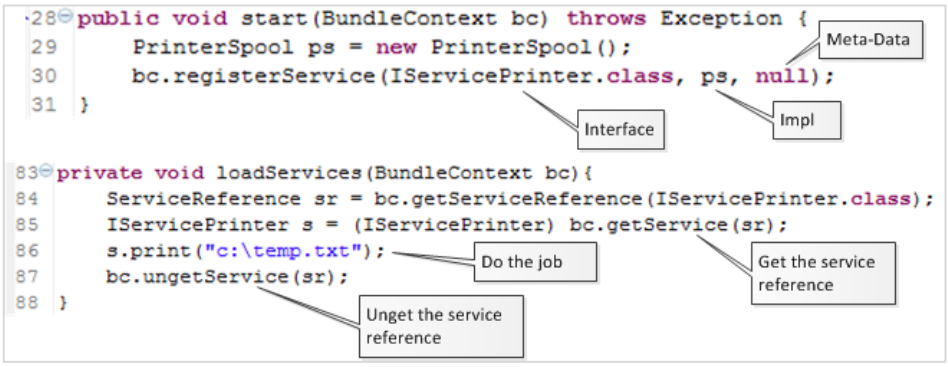



**Figura 40 – Registro e descoberta de Serviços**

Na linha 84 da **Figura 40**, é obtido o objeto de referência ao serviço (`ServiceReference`), e a partir desse é obtido o objeto de serviço (`IServicePrinter`) através do método `getService`, em seguida é realizado a chamada para o serviço (linha 86), e ao final, liberado a referência (linha 87). Conforme OSGi ALLIANCE SPECIFICATIONS (2011), um objeto do tipo `ServiceReference` encapsula as propriedades e meta-dados de um determinado objeto de serviço que ele representa.

Essas informações podem ser utilizadas para filtrar os serviços quando há vários fornecedores. Tais informações são configuradas no registro de cada serviço e filtradas na hora de obter a referência dos mesmos. Conforme HALL *et al*. (2010), o mecanismo utilizado para filtro se chama *LDAP Queries*.

Como pode ser observado na **Figura 41**, existem três blocos de códigos, dois referentes à *bundles* de serviços e o terceiro de um consumidor. O primeiro bloco registra um serviço, e atribui o valor 12 a propriedade `page-size`. Já o segundo bloco registra um serviço para a mesma interface e atribui o valor 10 a propriedade `page-size`. No terceiro bloco é realizada a pesquisa dos serviços utilizando o filtro. Ao executar esse código a consulta retornará apenas o serviço `PrinterService1`, caso ele esteja instalado e inicializado.

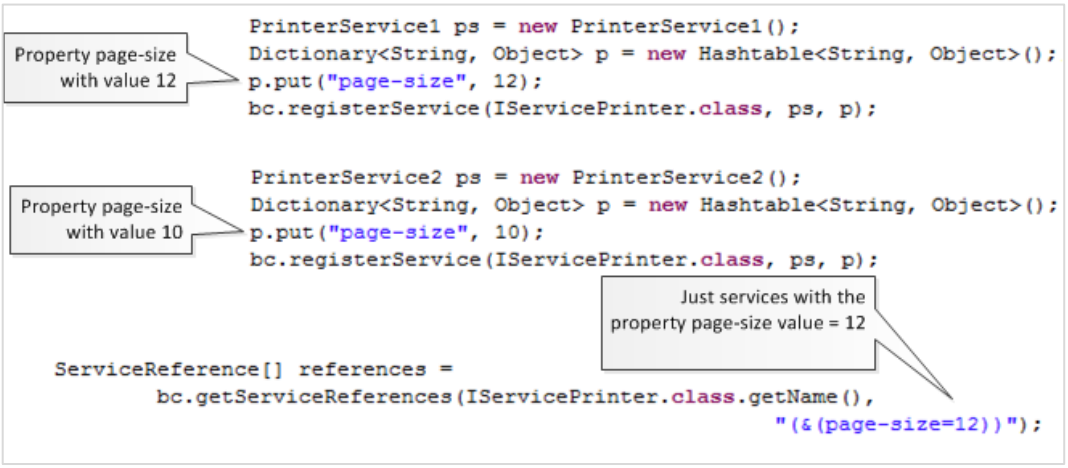



**Figura 41 – Pesquisa de serviços com filtros**

### 2.3.3.3.3 Quando usar serviços

A camada de serviços trás muito mais flexibilidade e dinamismo comparando se forem utilizadas apenas as camadas de modularização e ciclo de vida. Uma vez que ao atualizar um *bundle* B não é necessário resolver novamente o *bundle* A, a atualização é feita em tempo real, ou seja, na próxima tentativa de obter o serviço já estará publicada a nova versão.

Essa camada deve ser utilizada quando houver: a necessidade de flexibilidade e baixo acoplamento; softwares de alta disponibilidade e quando houver atualizações com frequência.

### 2.3.4 Implementações de referência

Conforme OSGi ALLIANCE SPECIFICATIONS (2011) como OSGi é um especificação, ela não fornece implementação, logo qualquer um pode desenvolver um *framework* baseado nele. Existem três implementações *open-source* conforme descritas abaixo:

- *Apache Felix*: projeto derivado do projeto Oscar, que foi utilizado para desenvolver a especificação. O qual foi escolhido para ser utilizado nesse trabalho;
- *Eclipse Equinox*: *framework* de base da IDE (*Integrated Development Environment*) eclipse;
- *Knopflerfish*: Projeto liderado e mantido pela empresa Makewave.

# 3 MATERIAIS E MÉTODOS

Para realização deste trabalho foi feita uma revisão bibliográfica sobre componentização. Um item utilizado nesse levantamento foi a breve aplicação de exemplos simples. Este capítulo tem por objetivo aplicar esses conhecimentos em exemplos semelhantes ou próximos da realidade.

## 3.1 METODOLOGIA DE TRABALHO

A metodologia de trabalho baseou-se em apresentar como aplicar e resolver os problemas descritos ao longo desse trabalho. Para tal serão desenvolvidos protótipos com finalidade de demonstração da solução dessas problemáticas. Tais protótipos serão apresentados no formato de diagrama de componentes e outras ilustrações gráficas. Artefatos de códigos fontes Java e arquivos de manifesto também são exibidos. Os componentes desenvolvidos tem fim demonstrativo, porém podem ser usados como exemplo para desenvolvimento comercial.

## 3.2 FERRAMENTAS UTILIZADAS

Para análise e desenvolvimento dos protótipos foram utilizadas as seguintes ferramentas:

- *Eclipse* 3.7 (*Indigo*). Essa ferramenta será utilizada com os fins de criação de código fonte Java. É uma ferramenta mantida pelo Eclipse Foundation e não tem fins lucrativos.
- *Astah Community 6.4.1*. Uma das ferramentas de licença livre da empresa *Astah.* O trata-se de uma ferramenta completa para geração de diagramas UML. Ela será utilizada para geração de diagrama de componentes.
- *Visio 2010*. Ferramenta de licença comercial fornecida pela *Microsoft* que será utilizada para geração de ilustrações simples;

- *Apache Felix 4.0.1*. Projeto implementação do *framework* OSGi de uso livre. Seu papel nesse trabalho será oferecer o ambiente de execução.

## 3.3 PROTÓTIPOS

Para cada um dos níveis de utilização do *framework* OSGi será apresentado um protótipo desenvolvido. Cada um destes tem por meta solucionar um ou mais problemas descritos ao longo desse trabalho.

## 3.4 PROTÓTIPO DE MODULARIZAÇÃO

Este protótipo tem por objetivo mostrar que é possível utilizar mais de uma versão da mesma classe no *classpath* da aplicação, sem o risco de no momento da execução da mesma ser chamada a classe que não deveria ser chamada. Outro objetivo também é garantir a visibilidade apenas interna de pacotes que não devem ser expostos aos demais *bundles*.

### 3.4.1 Análise e Implementação

A aplicação desse protótipo gera dois tipos de documentos *.pdf* e *.txt*. Existem dois módulos distintos que dependem dessa geração de documentos: *ModuleReport* e *ModuleTicket*. O *ModuleReport* é um módulo que gera relatórios, os mesmos devem ser disponibilizados no formato *pdf*, e por isso ele utiliza o *ModulePdf*. Já o *ModuleTicket* é responsável por preencher cupons fiscais realizadas nas compras. O conteúdo desses cupons é baseado em poucas informações e por isso elas são armazenadas em um arquivo *txt*. Para gerar tais arquivos existe a dependência com o módulo *ModuleTxt*, **Figura 42**.

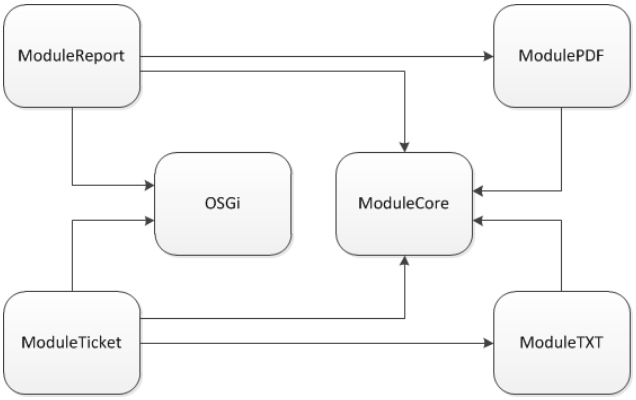



**Figura 42 – Protótipo de modularização**

A **Figura 43** apresenta a relação de dependência dos componentes apresentados na **Figura 42**. O componente OSGi nesse protótipo fornecerá a implementação da interface `BundleContext`, já a interface `BundleActivator` será implementada por classes dos componentes *ModuleReport* e *ModuleTicket*. O componente *ModuleCore* fornece a interface `IDocumentConverter` que define as operações para converter informações em arquivos. Os componentes *ModulePDF* e *ModuleTXT* fornecem a mesma classe `DocumentConverter`, que implementa a interface `IDocumentConverter`.

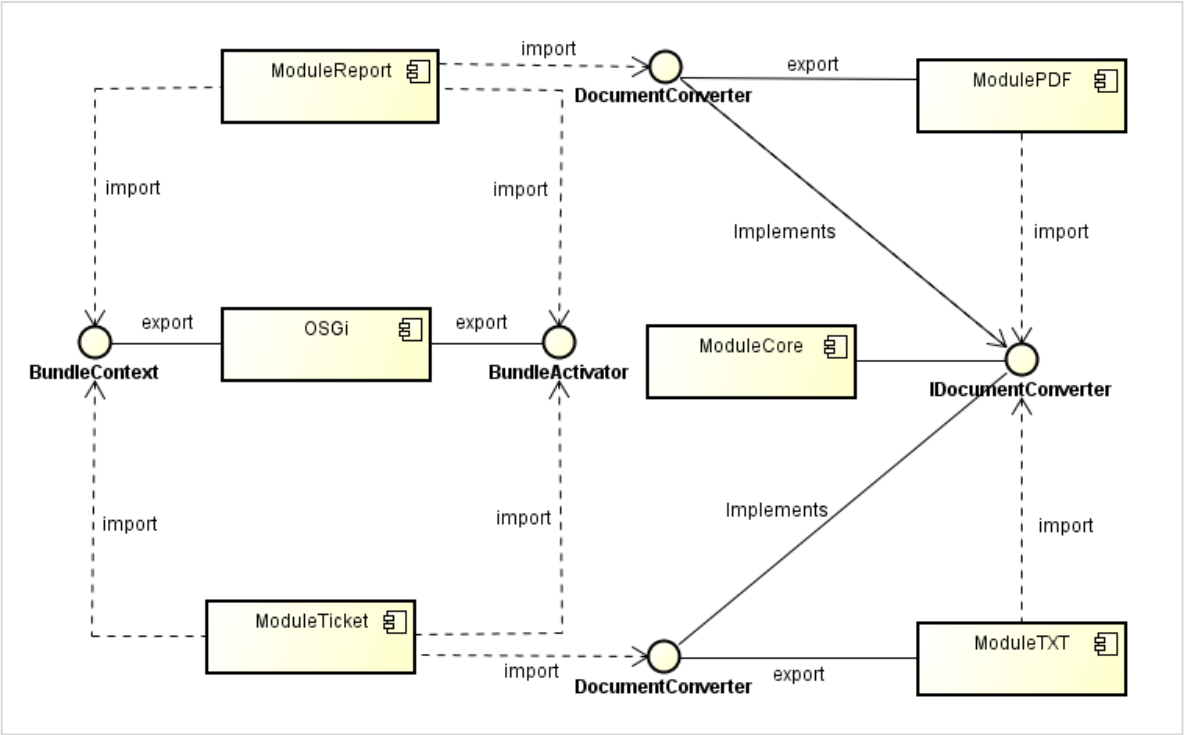



**Figura 43 – Diagrama de componentes do protótipo de modularização**

Na ferramenta *eclipse* foi criado um projeto Java padrão para cada um desses componentes, com exceção do OSGi, que é fornecida pela biblioteca compilada do *Apache Felix*. Como pode ser observado na **Figura 44** os projetos *ModuleTicket* e *ModuleReport* possuem dependência da biblioteca *Apache Felix*. Mais detalhes sobre os pacotes, classes e seus relacionamentos podem ser visualizados no **Apêndice A**.

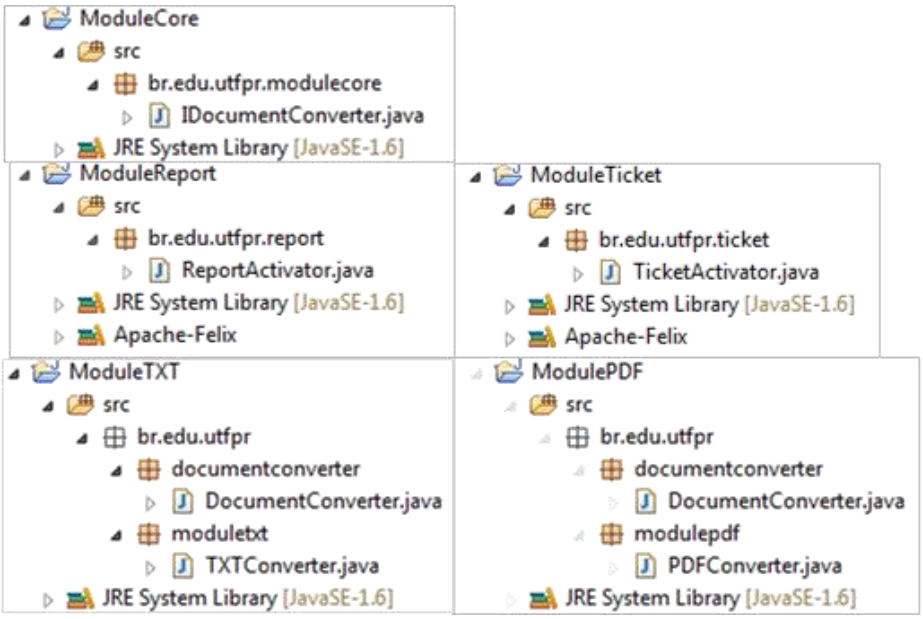



**Figura 44 – Projetos do protótipo de modularização**

Dentre todas essas classes destaca-se as duas versões da classe `DocumentConverter`, uma delas no projeto *ModulePDF* e a outra no *ModuleTXT*. Elas são exatamente a mesma classe quando se fala em nome de pacote e nome de classe, variando apenas em seus conteúdos. A **Figura 45** apresenta essa diferença, variando apenas no código do método `convertDocument()` nas linhas 10 e 12. Na primeira classe é realizado uma chamada para a classe `PDFConverter` e na outra para `TXTConverter`.

```
1  package br.edu.utfpr.documentconverter;
2
3  import java.io.File;
7
8  public class DocumentConverter implements IDocumentConverter {
9      public File convertDocument(String content) {
10         PDFConverter pdfConverter = new PDFConverter();
11         return pdfConverter.convertContentToPdf(content);
12     }
13 }
```

```
1  package br.edu.utfpr.documentconverter;
2
3  import java.io.File;
7
8  public class DocumentConverter implements IDocumentConverter {
9      public File convertDocument(String content) {
10         TXTConverter txtConverter = new TXTConverter();
11         return txtConverter.convertContentToTxt(content);
12     }
13 }
```

**Figura 45 – Duas classes idênticas na estrutura**

As classes `TicketActivator` e `ReportActivator` implementam a interface `BundleActivator`, ou seja, quando seus respectivos *bundles* forem iniciados pelo *framework*, elas serão executadas. Nas implementações dessas classes há chamadas para os métodos `convertDocument()` das classes `DocumentConverter`.

O próximo passo é configurar os arquivos de manifesto. A **Figura 46** destaca as configurações mais importantes dos arquivos manifesto de todos os componentes desse protótipo. A linha 2 apresenta a exportação de pacote do *bundle ModulePDF* na versão 2.0.0. Já na linha 4 o mesmo pacote é exportado na versão 1.0.0 para o *bundle ModuleTXT*. Na linha 6 e 7 é realizado a importação do pacote da versão 2.0.0 para o *bundle ModuleReport*. E nas linhas 9 e 10 é configurado a importação do pacote para o *bundle ModuleTicket*, restringindo essa importação entre as versões maiores e iguais a 1 e menores que 2. Com isso, o *bundle* do *pdf* não será importado para o *ModuleTicket*. Isso é necessário porque quando o *framework* resolve as dependências ele da prioridade aos pacotes de maiores versões.

```
#ModulePDF
Export-Package: br.edu.utfpr.documentconverter;version="2.0.0"
#ModuleTXT
Export-Package: br.edu.utfpr.documentconverter;version="1.0.0"
#moduleReport
Import-Package: org.osgi.framework, br.edu.utfpr.documentconverter;ver
 sion="2.0.0", br.edu.utfpr.modulecore
#ModuleTicket
Import-Package: org.osgi.framework, br.edu.utfpr.documentconverter;ver
 sion="[1, 2)", br.edu.utfpr.modulecore
```

**Figura 46 – Configurações importantes do arquivo de manifesto**

Todas as demais configurações dos arquivos de manifesto podem ser visualizadas no **Apêndice B**.

## 3.5 PROTÓTIPO DE CICLO DE VIDA

O protótipo de ciclo de vida (*Lifecycle*) tem por objetivo mostrar que é possível adicionar, atualizar e remover componentes em um sistema em tempo de execução. Foi desenvolvido um módulo principal onde o mesmo receberá a referência de outros módulos. O módulo principal será uma janela, onde nela para cada módulo inicializado no *framework*, haverá um botão que quando acionado, para irá acessar as funcionalidades internas do respectivo módulo.

### 3.5.1 Análise e implementação

Os componentes desenvolvidos nesse protótipo podem ser visualizados na **Figura 47**. Basicamente tem-se o *bundle* do *framework* OSGi, logo abaixo o há o *LifeCycleCore* que tema responsabilidade de disponibilizar um mecanismo para que os módulos ao serem iniciados e finalizados no OSGi, façam o mesmo no *LifeCycleCore*. Os outros dois módulos desenvolvidos são: *LifeCycleAdm* e *LifeCycleFinance*.

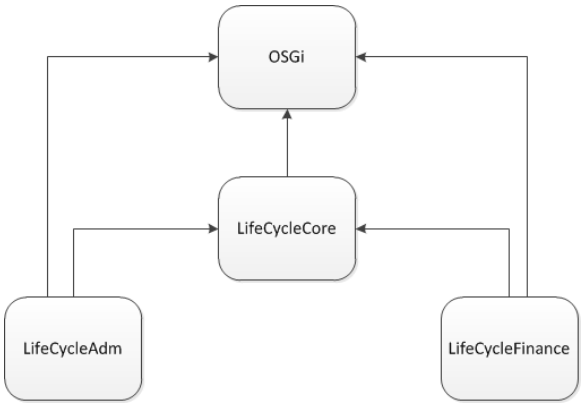



**Figura 47 – Protótipo de ciclo de vida**

O relacionamento entre os componentes bem como suas interfaces exportadas pode ser visualizado na **Figura 48**. Detalhe para o componente *LifeCycleCore* que exporta a interface `IModule` e os componentes *LifeCycleFinance* e *LifeCycleAdm* a implementam. Outra classe que também é exportada é a `FrameCore`, que possui as funcionalidades disponíveis aos módulos que implementem `IModule` se iniciem e finalizem no componente *LifeCycleCore*.

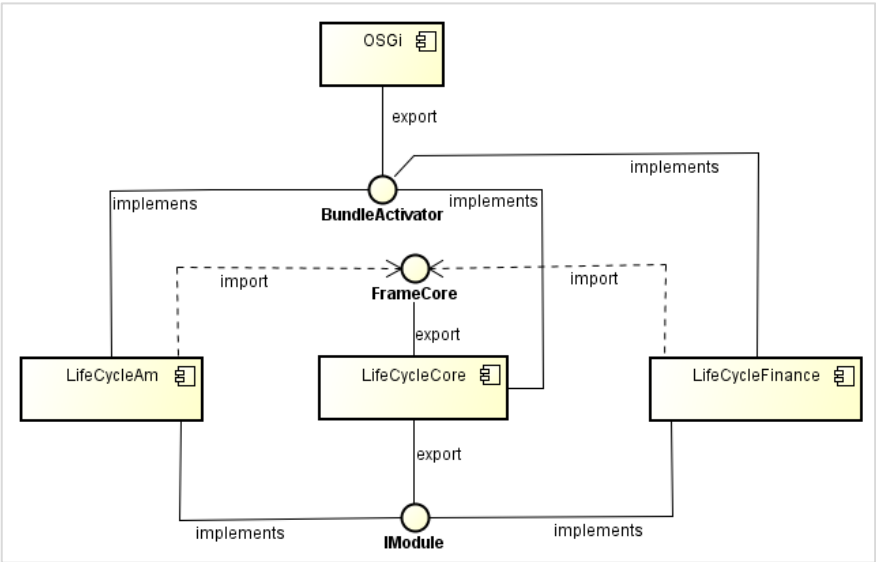



**Figura 48 – Diagrama de componentes do protótipo de ciclo de vida**

Definido a arquitetura do protótipo, os componentes são desenvolvidos no *eclipse* conforme pode ser visualizado na **Figura 49**.

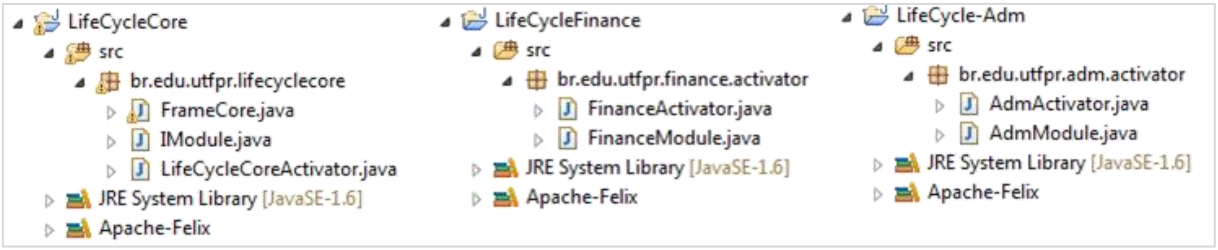



**Figura 49 – Projetos do protótipo de ciclo de vida**

Dentre todas essas classes destaca-se os métodos definidos na interface `IModule`, e os métodos da classe `FrameCore`. Na interface `IModule` é definido os métodos `open()` e `getBundle()`. Na classe `FrameCore` há o método `addModule()` que recebe uma instância de `IModule`. Através das informações desse objeto é adicionado um botão na tela, ao clicar nesse botão é disparado uma ação ao objeto `IModule`, através do método `open()`. Ainda no `FrameCore` há o método `removeModule()`, que remove o botão da tela quando o respectivo módulo estiver parado. Ao iniciar e finalizar os módulos, os mesmos deverão fornecer um objeto `IModule` através dos métodos da classe `FrameCore`, **Figura 50**.

Methods of IModule

```
5  public interface IModule {
6      void open();
7      Bundle getBundle();
8  }
```

Methods of FrameCore

```
67     public void addModule(final IModule module){
68         Bundle bundle = module.getBundle();
69         JButton b = new JButton(bundle.getHeaders().get("Bundle-Name")
70                                 + " "+ bundle.getVersion() );
71         panel.add(b);
72         b.addActionListener(new ActionListener() {
73             public void actionPerformed(ActionEvent arg0) {
74                 module.open();
75             }
76         });
77         mapModules.put(bundle, b);
78         panel.updateUI();
79     }
80     public void removeModule(IModule module){
81         panel.remove(mapModules.get(module.getBundle()));
82         panel.updateUI();
83     }
84 }
```

**Figura 50 – Interface *IModule* e classe *FrameCore***

## 3.6 PROTÓTIPO DE SERVIÇOS

O objetivo desse protótipo é demonstrar que é possível desenvolver componentes como serviços. Comparando com o nível de ciclo de vida, os serviços trazem maior disponibilidade, pois para trocar de um serviço para outro não é necessário parar os *bundles* que a ele fazem referência.

Será apresentado um protótipo de envio de Nota Fiscal Eletrônica (NFe). Haverá dois serviços que vão receber objetos do tipo `NFe`, *ServiceSefazMT* e *ServiceSefazSP*. A primeira opção de serviço sempre será MT, porém, caso a mesma esteja fora do ar deverá ser utilizado outro serviço, SP.

### 3.6.1 Análise e implementação

Esse protótipo será composto por quatro componentes: *ServiceCore*, *ServiceNFe*, *ServiceSefazMT* e *ServiceSefazSP*. *ServiceNFe* será responsável por criar os objetos `NFe`, e também por pesquisar o serviço adequado para enviar tais objetos. *ServiceSefazMT* e *ServiceSefazSP* serão os componentes que publicarão tais serviços no registro de serviços do *framework.*

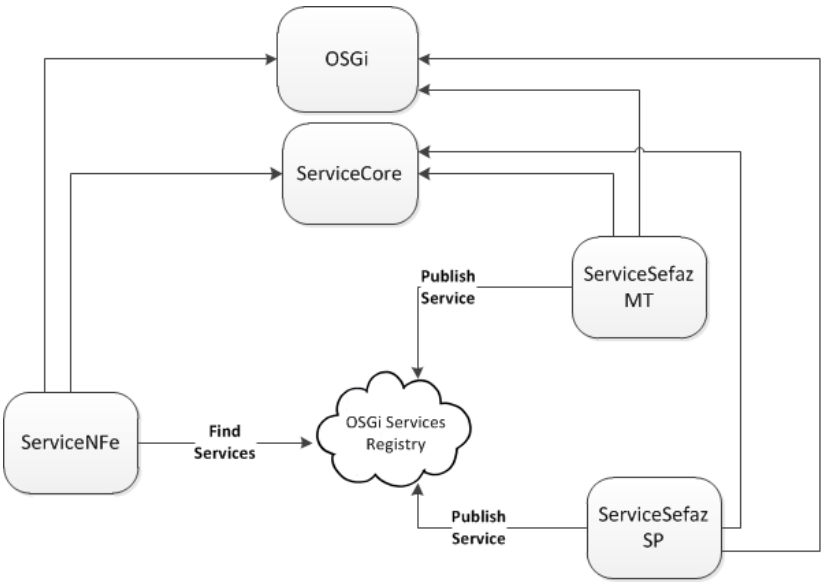



**Figura 51 – Protótipo de serviços**

Definido os componentes, os mesmos são desenvolvidos no *eclipse* conforme **Figura 52**. O projeto `ServiceCore` possui a interface `ISefazSender` que define as operações que um serviço de envio da `NFe` deve possuir. No mesmo projeto ainda há a classe `NFe` que encapsula as informações de uma nota fiscal. Nos projetos *ServiceSefazSP* e *ServiceSefazMT* há as seguintes classes: `ServiceActivator`, classe que será executada quando o *bundle* for inicializado. É nessa classe que os serviços serão publicados; `ServiceSefazImpl`, classe que implementa a interface `ISefazSender`; `ServiceConsole`, uma tela que exibe as informações recebidas do serviço. Já no projeto `ServiceNFE` há a classe `NFeActivator` que quando inicializada, exibirá a janela `FrameNFe`. Esta por sua vez é a classe responsável por fazer a pesquisa de serviços para envio da `NFe`.

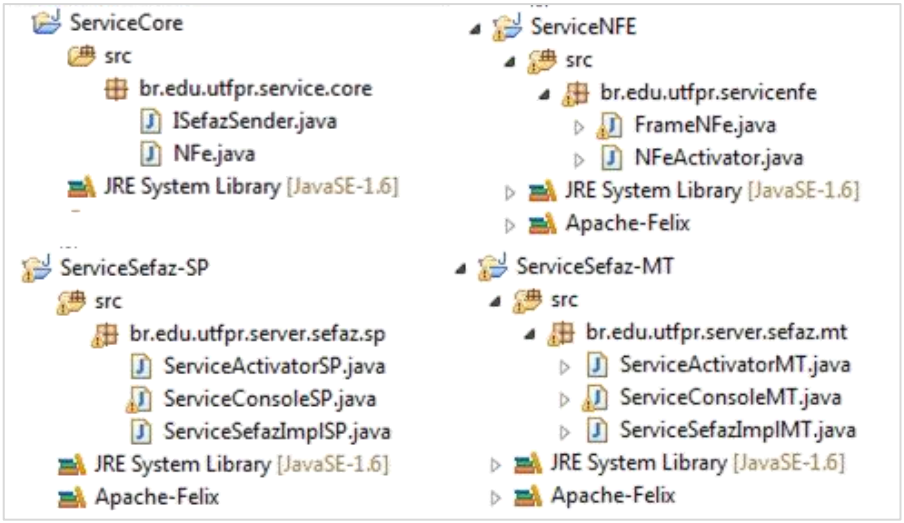



**Figura 52 – Projetos do protótipo de serviço**

As classes `ServiceSefazImplSP` e `ServiceSefazImplMT` registram o mesmo serviço. Para permitir que haja a possibilidade de livre escolha entre um serviço e outro, no momento da publicação são adicionadas propriedades no dicionário de dados de cada serviço. A propriedade a ser configurada nesse protótipo é '*uf*', `ServiceSefazImplSP` utilizará o valor 'SP', já `ServiceSefazImplMT` o valor 'MT', **Figura 53**.

```
13⊖     public void start(BundleContext bc) throws Exception {
14          ServiceSefazImplSP service = new ServiceSefazImplSP();
15          Dictionary<String, Object> d = new Hashtable<String, Object>();
16          d.put("uf", "SP");
17          bc.registerService(ISefazSender.class, service, d);
18      }
13⊖     public void start(BundleContext bc) throws Exception {
14          ServiceSefazImplMT service = new ServiceSefazImplMT();
15          Dictionary<String, Object> d = new Hashtable<String, Object>();
16          d.put("uf", "MT");
17          bc.registerService(ISefazSender.class, service, d);
18      }
```

**Figura 53 – Registro serviços com propriedades**

Por outro lado o método `startProcess()` da classe `FrameNFe` realiza a pesquisa dos serviços. Na linha 83 da **Figura 54** é realizada uma pesquisa pelos serviços que possuem a propriedade '*uf*' com o valor 'MT'. Caso não haja nenhum serviço com essa característica, o código tentará obter o primeiro serviço que ele encontrar (linha 92 a 94). Caso a referência do serviço seja obtida, nas linhas 98 e 99 a `NFe` é enviada, e o serviço é liberado na linha 100.

```
81⊖     public void startProcess(){
82          try {
83              ServiceReference[] references
84                  = bundleContext.getServiceReferences(ISefazSender.class.getName(),
85                                                       "(&(uf=MT))");
86              ISefazSender servico = null;
87              ServiceReference sr = null;
88              if (references != null && references.length >0){
89                  sr = references[0];
90                  servico = (ISefazSender) bundleContext.getService(sr);
91              }else {
92                  sr = bundleContext.getServiceReference(ISefazSender.class);
93                  if (sr != null){
94                      servico = (ISefazSender) bundleContext.getService(sr);
95                  }
96              }
97              if (servico != null){
98                  NFe nfe = getNfe();
99                  String retorno = servico.sendNfe(nfe);
100                 bundleContext.ungetService(sr);
101             }
102         } catch (Exception e) {
103             e.printStackTrace();
104         }
105     }
```

**Figura 54 – Código de pesquisa de serviços**

# 4 TESTES E RESULTADOS

Este capítulo tem por objetivo auditar os protótipos desenvolvidos no capítulo 3. Para o protótipo de modularização os testes são mais simples, pois é necessário apenas validar se foi possível utilizar as duas versões da mesma classe. Já no protótipo de ciclo de vida foi aplicados mais testes de comandos enviados para o *framework* durante sua execução. E o protótipo de serviços audita a alta disponibilidade e flexibilidade na troca dos mesmos.

## 4.1 PROTÓTIPO DE MODULARIZAÇÃO

Para esse protótipo será realizada apenas a inicialização do sistema com o propósito de verificar se duas classes com o mesmo nome e pacote podem ser referenciadas no *classpath*. Isso será possível através da versão do pacote. Para melhor visualizar os resultados foram adicionados no código alguns comandos para exibir mensagens na console. Tais códigos podem ser visualizados na **Figura 55**.

```
 9 public class TicketActivator implements BundleActivator{
10
11     public void start(BundleContext bc) throws Exception {
12         String content = "This is a module layer example";
13         IDocumentConverter idc = new DocumentConverter();
14         System.out.println("TicketActivator - file name = "+
15                     idc.convertDocument(content).getName());
16     }
```

```
 9 public class ReportActivator implements BundleActivator{
10
11     public void start(BundleContext bc) throws Exception {
12         String content = "This is a module layer example";
13         IDocumentConverter idc = new DocumentConverter();
14         System.out.println("ReportActivator - file name = "+
15                     idc.convertDocument(content).getName());
16     }
```

**Figura 55 – *TicketActivator* e *ReportActivator***

Com todos os *bundles* do protótipo copiados para a pasta *bundle* do *Apache Felix*, o mesmo é iniciado através do comando da linha 1 da **Figura 56**. Nas linhas 2 e 3 já é possível visualizar que o protótipo foi executado com sucesso, ou seja, as duas classes `DocumentActivator` foram executadas sem uma interferir na outra.

```
1  E:\Felix\felix-framework-4.0.1>java -jar bin/felix.jar
2  ReportActivator - file name = file.pdf
3  TicketActivator - file name = file.txt
4  ____________________________
5  Welcome to Apache Felix Gogo
6
7  g! lb
8  START LEVEL 1
9     ID|State      |Level|Name
10     0|Active     |    0|System Bundle (4.0.1)
11     1|Active     |    1|Module Core (1.0.0)
12     2|Active     |    1|Module Pdf (1.0.0)
13     3|Active     |    1|Module Report (1.0.0)
14     4|Active     |    1|Module Ticket (1.0.0)
15     5|Active     |    1|Module Txt (1.0.0)
16     6|Active     |    1|Apache Felix Bundle Repository (1.6.6)
17     7|Active     |    1|Apache Felix Gogo Command (0.12.0)
18     8|Active     |    1|Apache Felix Gogo Runtime (0.10.0)
19     9|Active     |    1|Apache Felix Gogo Shell (0.10.0)
20 g! stop 0
21 g! finishing TicketActivator!
22 Finishing ReportActivator!
```

**Figura 56 – Resultado do protótipo de modularização**

## 4.2 PROTÓTIPO DE CICLO DE VIDA

Para esse protótipo será seguido o seguinte script de teste:

1. Iniciar o *Apache Felix* com nenhum *bundle* do protótipo;
2. Instalar os três *bundles* nessa sequência: *LifeCycleCore*, *LifeCycleFinance* e *LifeCycleAdm*;
3. Iniciar os três *bundles* nessa sequência: *LifeCycleCore*, *LifeCycleFinance* e *LifeCycleAdm*;
4. Atualizar a versão 1.0.0 do *bundle LifeCycleFinance* para a versão 1.0.1;
5. Parar o *bundle LifeCycleAdm*;

#### 4.2.1.1 Iniciar o *Apache Felix*

Através do comando da linha 1 da **Figura 57** foi iniciado o *Apache Felix*. Com o mesmo em execução, foi executado o comando `lb` (*list bundles*) para listar todos os *bundles* instalados. Como pode ser observado, nenhum dos *bundles* do protótipo estão instalados, pode-se então passar para o próximo teste.

```
1  E:\Felix\felix-framework-4.0.1>java -jar bin/felix.jar
2
3  Welcome to Apache Felix Gogo
4
5  g! lb
6  START LEVEL 1
7     ID|State      |Level|Name
8      0|Active     |    0|System Bundle (4.0.1)
9      1|Active     |    1|Apache Felix Bundle Repository (1.6.6)
10     2|Active     |    1|Apache Felix Gogo Command (0.12.0)
11     3|Active     |    1|Apache Felix Gogo Runtime (0.10.0)
12     4|Active     |    1|Apache Felix Gogo Shell (0.10.0)
13 g!
```

**Figura 57 – Iniciando o *Apache Felix* do protótipo de ciclo de vida**

#### 4.2.1.2 Instalação dos *bundles* do protótipo

Com o *Apache Felix* em execução foram instalados os três *bundles* desenvolvidos para esse protótipo. Como pode ser observado na linha 1 da **Figura 58**, é executado o comando de instalação do *bundle* `LifeCycleCore-1.0.0.jar`. Na linha 2 o *framework* exibiu o ID atribuído ao mesmo. O mesmo procedimento foi feito para os demais *bundles* (linha 3 e linha 5) onde os mesmos receberam os IDs 6 e 7 respectivamente. Na linha 7 foi executado o comando para listar os *bundles* instalados, onde da linha 15 até a linha 17 é possível visualizar os *bundles* do protótipo corretamente instalados.

```
g! install file:e:/bundles/LifeCycleCore-1.0.0.jar
Bundle ID: 5
g! install file:e:/bundles/LifeCycleFinance-1.0.0.jar
Bundle ID: 6
g! install file:e:/bundles/LifeCycleAdm-1.0.0.jar
Bundle ID: 7
g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (4.0.1)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.6)
    2|Active     |    1|Apache Felix Gogo Command (0.12.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.10.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.10.0)
    5|Installed  |    1|LifeCycle Core (1.0.0)
    6|Installed  |    1|LifeCycle Finance (1.0.0)
    7|Installed  |    1|LifeCycle Adm (1.0.0)
g!
```

**Figura 58 – Resultado da instalação dos *bundles***

#### 4.2.1.3 Inicialização dos *bundles*

Primeiramente deve se iniciar o componente *LifeCycleCore* do protótipo, pois o mesmo receberá a referência dos demais *bundles* quando os mesmos forem inicializados. Para iniciar o mesmo, na console digita-se o comando `start 5`, que é o ID desse *bundle*. O resultado pode ser visualizado na **Figura 59**, onde não há nenhum módulo inicializado.


**Figura 59 – *LifeCycleCore* iniciado e vazio**

Em seguida é iniciado o *bundle LifeCycleFinance* através do comando `start 6`, que é o ID atribuído a ele. O *bundle LifeCycleAdm* também foi iniciado através do comando `start 7`. A **Figura 60** apresenta os dois módulos devidamente inicializados. Cada um possui seu respectivo botão na tela principal do protótipo. No lado direito da ilustração é ilustradas as janelas exibidas na execução de cada botão. Os códigos que exibem tais janelas, estão contidos nas classes de implementação da interface `IModule`, **Figura 50.**

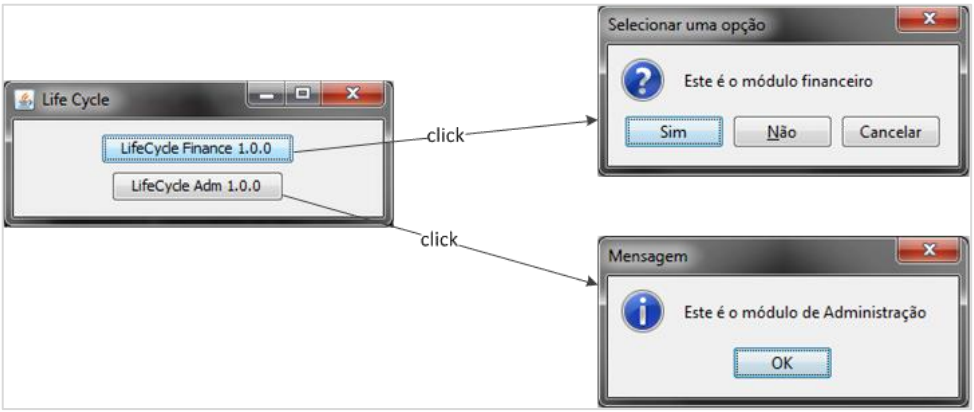



**Figura 60 – *Bundles* do protótipo inicializados**

#### 4.2.1.4 Atualização de *bundle*

Para demonstrar como atualizar um *bundle* sem parar a aplicação, foi alterada uma das classes do *bundle LifeCycleFinance*. Tal alteração mudou a forma de como é exibida a mensagem na tela, quando for executado o botão da tela principal do protótipo. Na versão 1.0.0 a mensagem é exibida em uma caixa de pergunta, agora será exibida uma caixa de informação.

Depois de realizada a alteração, a aplicação foi compilada e empacotada em um novo pacote na versão 1.0.1. Para atualizar o *bundle* foi executado o comando do **Quadro 7**.

```
g! update 6 file:e:/bundles/LifeCycleFinance-1.0.1.jar
```

**Quadro 7 – Atualização do *bundle* do protótipo**

Ao executar esse comando, o *framework* parou o *bundle*, logo o mesmo foi removido da tela. Em seguida o *framework* internamente atualizou as definições de classes e reiniciou o *bundle*. Ao ser iniciado o mesmo foi adicionado novamente na tela do protótipo. Ao clicar no botão *LifeCycleFinance* 1.0.1 é exibida a caixa de mensagem de informação atualizada, **Figura 61.**

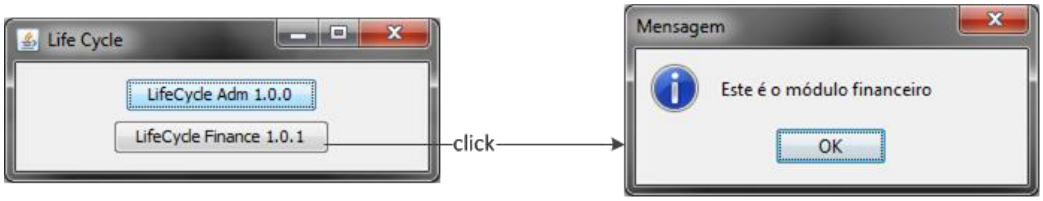



**Figura 61 – *Bundle* atualizado**

#### 4.2.1.5 Remoção de *bundle*

Da mesma maneira que um *bundle* ao ser iniciado deverá ser adicionado um botão na tela do principal do protótipo, ao ser parado o botão deverá ser removido, pois o respectivo *bundle* está desinstalado ou parado. Para testes será parado o *bundle LifeCycleAdm* através do comando da linha 1 da **Figura 62**. Ao listar os *bundles* (linha 2) percebe-se que o mesmo está com o estado de resolvido (linha 12).

```
1  g! stop 7
2  g! lb
3  START LEVEL 1
4     ID|State      |Level|Name
5      0|Active     |    0|System Bundle (4.0.1)
6      1|Active     |    1|Apache Felix Bundle Repository (1.6.6)
7      2|Active     |    1|Apache Felix Gogo Command (0.12.0)
8      3|Active     |    1|Apache Felix Gogo Runtime (0.10.0)
9      4|Active     |    1|Apache Felix Gogo Shell (0.10.0)
10     5|Active     |    1|LifeCycle Core (1.0.0)
11     6|Active     |    1|LifeCycle Finance (1.0.1)
12     7|Resolved   |    1|LifeCycle Adm (1.0.0)
13 g!
14
15
16
17
18
19
20
```

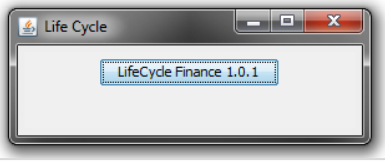



**Figura 62 – Parando um *bundle***

## 4.3 PROTÓTIPO DE SERVIÇOS

Para esse protótipo será executado o seguinte script de teste:

1. Iniciar o módulo *ServiceNFe* sem os demais serviços iniciados: a código ficará em um laço de repetição tentando obter uma referência do serviço;
2. Iniciar os dois serviços: o consumidor deverá chamar o serviço conforme prioridade definida;
3. Ao parar o serviço que tem prioridade, o outro que não atendia o critério de escolha deverá ser utilizado.

Para os testes desse protótipo será considerado que o *ServiceCore* estará na pasta *bundle* do *Apache Felix*, e quando o *framework* for iniciado o mesmo será instalado e inicializado automaticamente.

#### 4.3.1.1 Iniciar o *ServiceNFe* sem os serviços iniciados

Primeiramente inicia-se o *framework* através da linha 1 da **Figura 63.** Após isso é instalado o serviço *ServiceNFe* (linha 5) e iniciado na sequência (linha 6). O resultado pode ser observado na janela *Service NFe – Console*, onde o consumidor não encontra nenhum serviço disponível.

```
1  E:\Felix\felix-framework-4.0.1>java -jar bin/felix.jar
2  ____________________________
3  Welcome to Apache Felix Gogo
4
5  g! install file:e:/bundles/ServiceNfe-1.0.0.jar
6  Bundle ID: 6
7  g! start 6
8  g!
```

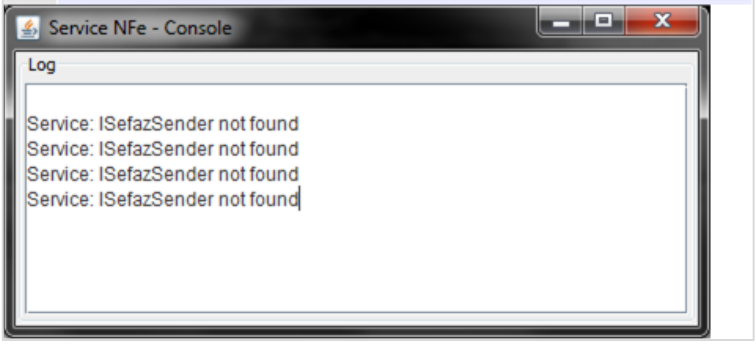



**Figura 63 – Iniciando o consumidor sem os serviços**

#### 4.3.1.2 Iniciar os dois serviços

Com o consumidor em execução é instalado e iniciado os dois serviços. Na linha 1 é instalado o serviço de MT e na linha 3 o de SP. Nas linhas 5 e 6 os dois são iniciados. Na linha 7 foi executado o comando para listar os *bundles* instalados, percebe-se que os dois serviços estão ativos (linhas 16 e 17), **Figura 64**.

```
1  g! install file:e:/bundles/ServiceSefazMT-1.0.0.jar
2  Bundle ID: 7
3  g! install file:e:/bundles/ServiceSefazSP-1.0.0.jar
4  Bundle ID: 8
5  g! start 7
6  g! start 8
7  g! lb
8  START LEVEL 1
9     ID|State      |Level|Name
10     0|Active     |    0|System Bundle (4.0.1)
11     1|Active     |    1|Apache Felix Bundle Repository (1.6.6)
12     2|Active     |    1|Apache Felix Gogo Command (0.12.0)
13     3|Active     |    1|Apache Felix Gogo Runtime (0.10.0)
14     4|Active     |    1|Apache Felix Gogo Shell (0.10.0)
15     6|Active     |    1|ServiceNFe (1.0.0)
16     7|Active     |    1|Service SEFAZ MT (1.0.0)
17     8|Active     |    1|Service SEFAZ SP (1.0.0)
18    10|Active     |    1|Service Core (1.0.0)
19 g!
```

**Figura 64 – Iniciando os dois serviços da Sefaz**

Na próxima iteração do consumidor o mesmo já fará referência para o serviço iniciado. O resultado pode ser visualizado na janela do consumidor: *Service NFe – Console* e a janela do serviço: *Sefaz MT – Console*, **Figura 65.**

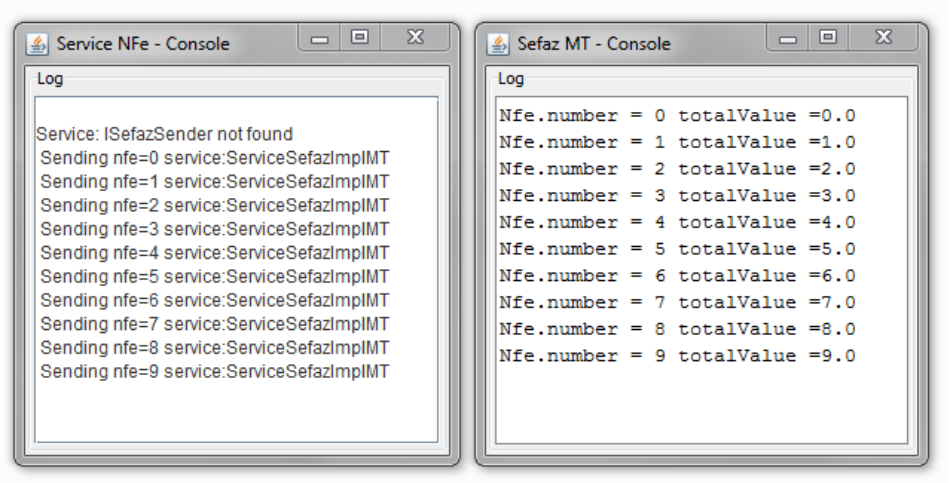



**Figura 65 – Console do consumidor e serviço MT**

#### 4.3.1.3 Parando um dos serviços

O último teste realizado é parar o serviço de preferência (MT), através do comando `stop 7`, onde 7 é o ID do *bundle ServiceSefazMT*. Na iteração seguinte do código do componente *ServiceNFe*, passou-se a utilizar a referência do outro serviço, *ServiceSefazSP*. O resultado pode ser visualizado na **Figura 66.**

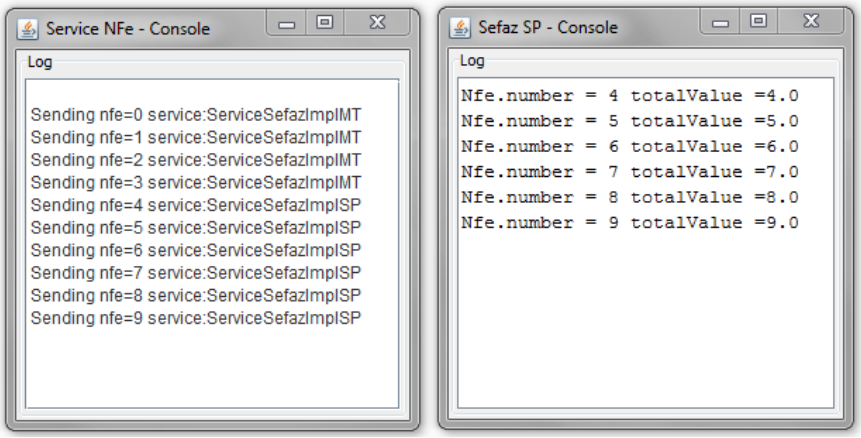



**Figura 66 – Troca de serviços**

Observa-se na **Figura 66** que após a paralização do serviço *ServiceSefazMT*, passou a ser utilizado o serviço ServiceSefazSP.

# 5 CONSIDERAÇÕES FINAIS

O presente capítulo apresenta as considerações finais, tanto conclusões quanto considerações para trabalhos futuros, do estudo apresentado neste trabalho.

## 5.1 CONCLUSÕES

Conclui-se, através dos resultados obtidos, que o desenvolvimento baseado em componentes mostrou-se muito produtivo. Projetar um sistema dividindo-o em componentes/módulos traz mais clareza na percepção das funcionalidades, bem como da interação entre elas.

O *framework* OSGi se apresentou como uma ótima ferramenta, pois se encaixou perfeitamente em todos os requisitos do desenvolvimento baseado em componentes. Em aplicativos Java é possível desenvolver componentes de alta coesão, pois componentes possuem suas funcionalidades bem definidas. Os mesmos podem ser de baixo acoplamento, pois as interconexões são restritas a pacotes e interfaces. Sua arquitetura permite trocar determinados componentes do sistema sem afetar os demais.

Foi possível atender mecanismos não fornecidos nativamente pela linguagem Java, como por exemplo: conseguir usar na mesma aplicação duas versões da mesma classe; em nível de ciclo de vida é possível instalar, atualizar, remover componentes em tempo de execução; e melhor ainda, no nível de serviços é possível utilizar componentes dinamicamente, realizar a troca entre um serviço e outro, sem a necessidade de parar os componentes consumidores para isso.

## 5.2 SUGESTÕES PARA TRABALHOS FUTUROS

Após o término desse trabalho foram levantadas várias recomendações para trabalhos futuros, conforme listadas a seguir:

- Recomenda-se o estudo utilizando o *framework* OSGi voltado para aplicações empresariais utilizando o *Eclipse Virgo*. Essa ferramenta oferece uma plataforma simples, porém ampla para desenvolvimento, onde é possível implantar aplicações e serviços corporativos em Java (JEE). Esse *framework* define o conceito de WAB (*Web Application Bundle*) que é baseado no WAR (*Web Application Resource*). Essa ferramenta está atualmente na versão 3.0.
- Desenvolver um estudo comparativo entre as implementações do OSGi: *Apache Felix*, *Equinox* e *Knopflerfish*.
- Desenvolver um estudo avançado sobre o *framework* OSGi. Onde se encaixa a pesquisa sobre eventos, módulos dinâmicos, fragmentação de *bundles*, busca avançada de serviços explorando o LDAP Queries e aplicação de segurança utilizando a camada *Security Layer*.
- Integração do OSGi como *Apache Maven*, para geração de pacotes, repositório e download de dependências. Essa integração pode ser feita através do *Maven Bundle Plugin*.

# REFERÊNCIAS

ALLIANCE, O. OSGi Service Platform Core Specification Release 4 Version 4.3. **OSGi Alliance Specifications**, abr. 2011. Disponivel em: <http://www.osgi.org/Specifications/HomePage>.

BARBOSA, N. M.; PERKUSICH, A. **Estudo Experimental Comparativo de Modelos de Componentes para o Desenvolvimento de Software com Suporte à Evolução Dinâmica e não Antecipada**. Campina Grande-PB: [s.n.], 2006.

BLOIS, A. P. T. B.; WERNER, C. M. L.; BECKER, K. **Um processo de Engenharia de Dominio com foco no Projeto Arquitetural Baseado em Componentes**. João Pessoa, PB: [s.n.], 2004.

DELPIZZO, P. R. Alta disponibilidade em Portais Corporativos: quando implementar? **TecMedia**, 2008. Disponivel em: <http://www.tecmedia.com.br/novidades/artigos/alta-disponibilidade-em-portais-corporativos-quando-implementar>. Acesso em: 12 dez. 2011.

D'SOUZA, D.; WILLS, A. **Objects Components, and Frameworks with the UML:** The Catalysis approach. Boston: Addisson-Wesley, 1998.

GÉDÉON, W. **OSGi and Apache Felix 3.0**. Birmingham: Packt, 2010.

GIMENEZ, ITANA MARIA SOUZA DE; HUZITA, ELIZA HATSUE MORIYA. **Desenvolvimento Baseado em Componentes:** Conceitos e Técnicas. Rio de Janeiro: Ciência Moderna, 2005.

HALL, RICHARD S.; PAULS, KARL; MCCULLOCH, STUART; SAVAGE, DAVID. **OSGi in Action**. [S.l.]: Manning, 2010.

HEINEMAN, G. T. **An Evaluation of Component Adaptation**. Worcester, MA: Worcester Polytechnic Institute, 1999.

IEEE. **IEEE Standard Glossary of Software Engineering Terminology**. New York: [s.n.], 1990.

KRUEGER, C. Software Reuse. **ACM Computing Surveys**, New York, 24, n. 2, 1992.

NEIGHBORS, J. **Software Construction Using Components**. Universidade da Califórinia. Irvine - EUA. 1981.

ORACLE. Using JAR Files: The Basics. **Sun Developer Network**, 2010. Disponivel em: <http://java.sun.com/developer/Books/javaprogramming/JAR/basics/manifest.html>. Acesso em: 12 set. 2010.

PIETRO-DIAZ, R; ARANGO, GUILHERME. **Domain analysis and software systems modeling**. California: Ieee Computer Society , 1991.

RESENDE, ANA RUBÉLIA MENDES DE LIMA; CUNHA, ADILSON MARQUES DA; RESENDE, ANTONIO MARIA PEREIRA DE. **Um Modelo de Processo para seleção de componentes de Software**. [S.l.]: UFLA, 2007.

SAMETINGER, J. **Software Engineering with Reusable Components**. Berlin: Springer, 1997.

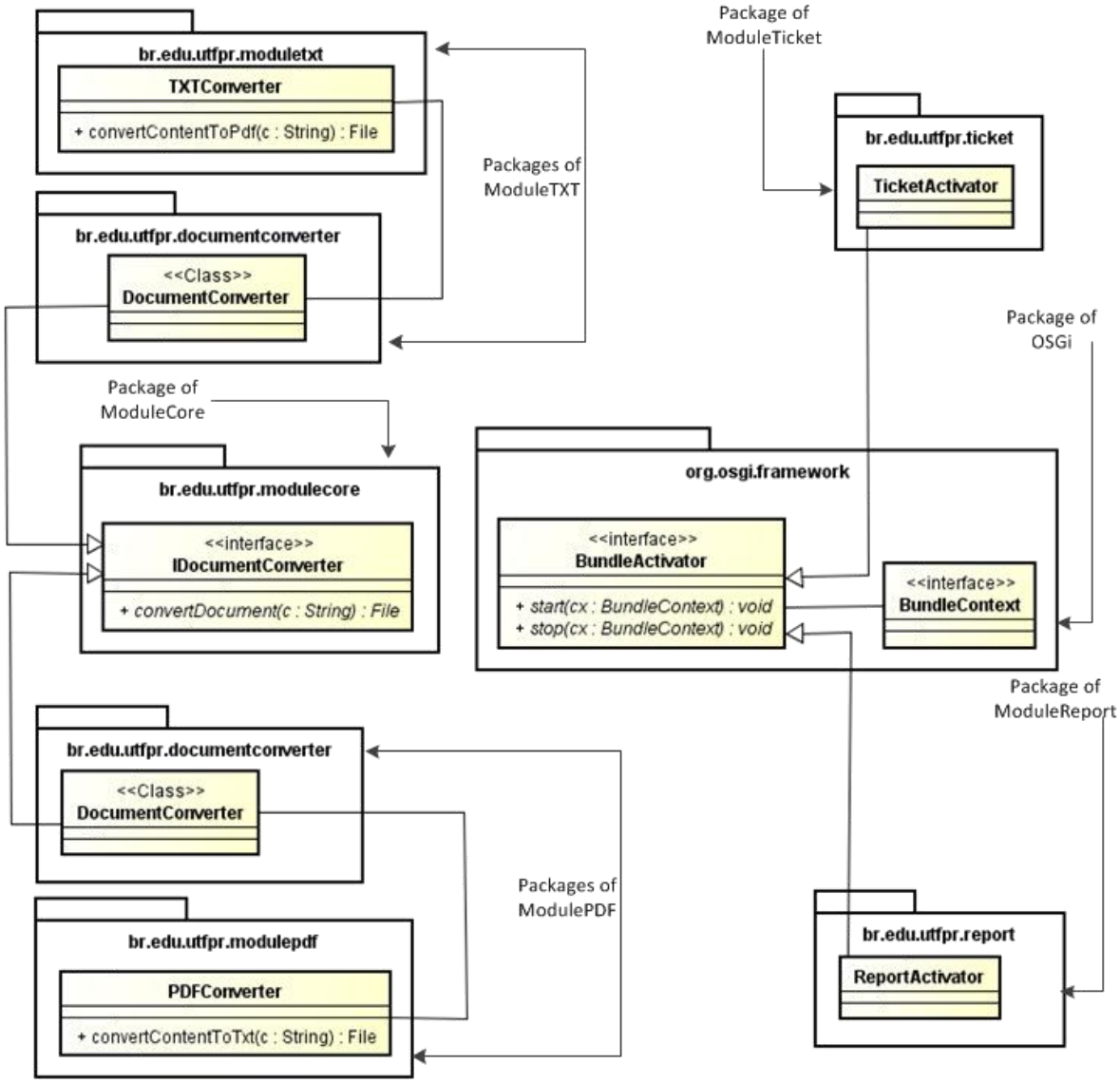
SIMOS, M. **Organization Domain Modeling (ODM):** Domain engineering as a co-methodology to object oriented techniques. Californica: Hewlett-Packard Laboratories, 1996.

THE OSGi Architecture. **OSGI Alliance**, 2011. Disponivel em: <http://www.osgi.org/About/WhatIsOSGi>. Acesso em: 18 jul. 2011.

VILLELA, R. M. M. B. **Busca e Recuperação de Componentes em Ambientes de Reutilização de Software**. UFRJ. Rio de Janeiro. 2000.

# APÊNDICES(S)

Apêndice A – Diagrama de pacotes do protótipo de modularização




Apêndice B – Arquivos de Manifesto do protótipo de modularização

```
#ModuleCore
Bundle-ManifestVersion: 2
Bundle-Name: Module Core
Bundle-Version: 1.0.0
Bundle-SymbolicName: br.edu.utfpr.modulecore
Export-Package: br.edu.utfpr.modulecore;version="1.0.0"

#ModulePDF
Bundle-ManifestVersion: 2
Bundle-Name: Module Pdf
Bundle-Version: 1.0.0
Bundle-SymbolicName: br.edu.utfpr.modulepdf
Import-Package: br.edu.utfpr.modulecore
Export-Package: br.edu.utfpr.documentconverter;version="2.0.0"


#ModuleTXT
Bundle-ManifestVersion: 2
Bundle-Name: Module Txt
Bundle-Version: 1.0.0
Bundle-SymbolicName: br.edu.utfpr.moduletxt
Import-Package: br.edu.utfpr.modulecore
Export-Package: br.edu.utfpr.documentconverter;version="1.0.0"

#ModuleReport
Bundle-ManifestVersion: 2
Bundle-Name: Module Report
Bundle-Version: 1.0.0
Bundle-SymbolicName: br.edu.utfpr.report
Bundle-Activator: br.edu.utfpr.report.ReportActivator
Import-Package: org.osgi.framework, br.edu.utfpr.documentconverter;ver
 sion="2.0.0", br.edu.utfpr.modulecore

 #ModuleTicket
 Bundle-ManifestVersion: 2
Bundle-Name: Module Ticket
Bundle-Version: 1.0.0
Bundle-SymbolicName: br.edu.utfpr.ticket
Bundle-Activator: br.edu.utfpr.ticket.TicketActivator
Import-Package: org.osgi.framework, br.edu.utfpr.documentconverter;ver
 sion="[1, 2)", br.edu.utfpr.modulecore
```