

**FELIPE RODRIGUES DO PRADO
JOÃO PAULO NAKAJIMA PEREIRA**

**DESENVOLVIMENTO MODULAR DE SOFTWARE
UTILIZANDO OSGI**

**UNIVERSIDADE DO VALE DO SAPUCAÍ
POUSO ALEGRE
2015**

**FELIPE RODRIGUES DO PRADO
JOÃO PAULO NAKAJIMA PEREIRA**

**DESENVOLVIMENTO MODULAR DE SOFTWARE
UTILIZANDO OSGI**

Trabalho de Conclusão de Curso apresentado
ao Curso de Sistemas de Informação da
Universidade do Vale do Sapucaí como
requisito parcial para obtenção do título de
bacharel em Sistemas de Informação

Orientador: Me. Márcio Emílio Cruz Vono de
Azevedo.

**UNIVERSIDADE DO VALE DO SAPUCAÍ
POUSO ALEGRE
2015**

LISTA DE FIGURAS

Figura 1 – Uso e reúso de partes do <i>software</i>	11
Figura 2 – Arquitetura de camadas.....	11
Figura 3 – Arquitetura modular OSGi.....	12
Figura 4 – Impacto das mudanças.....	13
Figura 5 – Estrutura de camadas.....	15
Figura 6 – Estados dos <i>bundles</i>	16
Figura 7 – OSGi atuando em diferentes setores.....	19
Figura 8 – Modelagem do banco de dados.....	36
Figura 9 – Arquitetura de um módulo do sistema.	37
Figura 10 – Representação dos serviços disponibilizados pelos módulos.....	38
Figura 11 – Representação dos módulos que consomem serviços.....	38
Figura 12 – Arquitetura dos módulos do sistema.....	39
Figura 13 – Opção para criar projeto.....	41
Figura 14 – Tela para escolha do tipo do projeto como POM Project.....	41
Figura 15 – Tela de informações do novo projeto.....	42
Figura 16 – Criação de novo projeto como módulo.....	43
Figura 17 – Estrutura do projeto principal composto por outro projeto.....	43
Figura 18 – Criação de novo projeto como módulo.....	44
Figura 19 – Tela para escolha do tipo do projeto como Web Application.....	44
Figura 20 – Tela de configuração do Server e Java EE Version.....	45
Figura 21 – Tela para escolha do tipo do projeto como Java Application ou OSGi Bundle....	45
Figura 22 – Tela para escolha do tipo do projeto como Enterprise JavaBeans.....	46
Figura 23 – Estrutura do projeto composta por um módulo do sistema.....	47
Figura 24 – Localização do arquivo pom.xml na estrutura do projeto.....	48
Figura 25 – Informações principais do arquivo pom.xml.....	48
Figura 26 – Informações do gerenciamento de dependências.....	50
Figura 27 – Configurações de <i>plugins</i> (parte 1).....	51
Figura 28 – Configurações de <i>plugins</i> (parte 2).....	52
Figura 29 – Configurações de <i>plugins</i> (parte 3).....	53
Figura 30 – Configurações de <i>plugins</i> (parte 4).....	53

Figura 31 – Arquivo <code>osgi.properties</code>	54
Figura 32 – Serviço do Módulo Log API.....	55
Figura 33 – Implementação do serviço do Módulo Log API.....	55
Figura 34 – Registro e remoção de serviços no <i>framework</i>	56
Figura 35 – Registro de serviços utilizando anotações da especificação EJB.....	57
Figura 36 – Instalação da ferramenta Apache Felix Web Console Bundles.....	58
Figura 37 – Inicialização do servidor de aplicação GlassFish.....	58
Figura 38 – Ferramenta Apache Felix Web Console Bundles.....	59
Figura 39 – Instalação de módulos através do Web Console.....	59
Figura 40 – Estrutura do projeto na IDE Netbeans.....	60
Figura 41 – Obtendo serviço do Módulo Log.....	61
Figura 42 – Tela principal do <i>software</i>	62
Figura 43 – Módulos ativos no menu do sistema.....	62
Figura 44 – Módulo API, <i>Core</i> e <i>View</i> do Módulo Clientes instalados e ativos.....	64
Figura 45 – Módulo de cadastro de clientes ativo no sistema.....	64
Figura 46 – Módulos <i>Core</i> e <i>View</i> do Módulo Clientes parados.....	65
Figura 47 – Módulo de cadastro de clientes parado no sistema.....	65
Figura 48 – Módulo <i>View</i> do cadastro de clientes parado no sistema.....	66
Figura 49 – Diagrama representando a granularidade do <i>software</i>	67
Figura 50 – Módulo Log instalado com a versão 1.0.0 e 2.0.0.....	69
Figura 51 – Trecho de código e dependências demonstrando o versionamento.....	69

LISTA DE ABREVIATURAS E SIGLAS

API	<i>Application Programming Interface</i>
ARO	<i>Army Research Office</i>
BSD	<i>Berkeley Software Distribution</i>
CDI	<i>Contexts and Dependency Injection</i>
CPEG	<i>Core Platform Expert Group</i>
CRUD	<i>Create, Retrieve, Update, Delete</i>
CSS	<i>Cascading Style Sheet</i>
DBA	<i>DataBase Administrator</i>
EEG	<i>Enterprise Expert Group</i>
EJB	<i>Enterprise JavaBeans</i>
EPL	<i>Eclipse Public License</i>
EUA	<i>Estados Unidos da América</i>
HTML	<i>HyperText Markup Language</i>
HTTP	<i>HyperText Transfer Protocol</i>
IDE	<i>Integrated Development Environment</i>
IoT	<i>Internet of Things</i>
JAR	<i>Java Archive</i>
Java EE	<i>Java Enterprise Edition</i>
JDK	<i>Java Development Kit</i>
JEE	<i>Java Platform, Enterprise Edition.</i>
JSON	<i>JavaScript Object Notation</i>
JVM	<i>Java Virtual Machine</i>
MEG	<i>Mobile Expert Group</i>
MVC	<i>Model, View, Controller</i>
NSF	<i>National Science Foundation</i>
NTT	<i>Nippon Telegraph and Telephone</i>
OSGi	<i>Open Services Gateway initiative</i>
POM	<i>Project Object Model</i>
REG	<i>Residential Expert Group</i>
REST	<i>Representational State Transfer</i>

SGBDR	Sistema Gerenciador de Banco de Dados Relacional
SQL	<i>Structured Query Language</i>
SRA	<i>Systems Research and Applications Corporation</i>
URI	<i>Uniform Resource Identifier</i>
URL	<i>Uniform Resource Locator</i>
VEG	<i>Vehicle Expert Group</i>
TDC	<i>The Developers Conference</i>
XHTML	<i>Extensible HyperText Markup Language</i>
XML	<i>Extensible Markup Language</i>
WAR	<i>Web Application Archive</i>
W3C	<i>World Wide Web Consortium</i>

SUMÁRIO

INTRODUÇÃO.....	7
2 QUADRO TEÓRICO.....	10
2.1 Modularização.....	10
2.2 Especificação OSGi.....	14
2.3 Java.....	20
2.4 Apache Felix.....	22
2.5 GlassFish.....	22
2.6 RESTful Web Services.....	23
2.7 Maven.....	25
2.8 PostgreSQL.....	26
2.9 HyperText Markup Language (HTML).....	27
2.10 JavaScript.....	27
2.11 Cascading Style Sheet (CSS).....	28
2.12 AngularJS.....	29
2.13 Bootstrap.....	29
3 QUADRO METODOLÓGICO.....	31
3.1 Tipo de pesquisa.....	31
3.2 Contexto de pesquisa.....	31
3.3 Instrumentos.....	32
3.4 Procedimentos.....	34
3.4.1 Prototipação.....	34
3.4.2 Definição do <i>software</i>	35
3.4.3 Modelagem do banco de dados.....	36
3.4.4 Modelagem da arquitetura dos módulos.....	37
3.4.5 Desenvolvimento.....	40
3.5 Resultados.....	60
4 DISCUSSÃO DE RESULTADOS.....	63
REFERÊNCIAS.....	71

INTRODUÇÃO

O desenvolvimento de um *software* não envolve apenas métodos de programação. Para desenvolvê-lo é necessário preocupar-se com planejamento, engenharia, metodologia e tecnologia a ser utilizada. Esses fatores influenciam na manutenção, atualização e expansão do mesmo. Dessa forma, definir tais fatores é fundamental para que o *software* seja flexível a mudanças. Pensando dessa maneira, será demonstrado um modelo de desenvolvimento modular, utilizando a especificação OSGi.

OSGi, abreviação para *Open Services Gateway Initiative*, possibilita o desenvolvimento de *softwares* em módulos, disponibilizando o gerenciamento dos mesmos e fornece facilidades na manutenção e expansão da aplicação.

Segundo Fernandes (2009), um sistema modular possui algumas propriedades. Deve ser autocontido, os módulos podem ser incluídos, retirados, instalados ou desinstalados. Outra propriedade é ter alta coesão. Um módulo deve cumprir apenas sua finalidade, ou seja, deve fazer somente as funções que lhe foram atribuídas. O baixo acoplamento é outra propriedade muito importante. Um módulo não precisa se preocupar com implementações de outros módulos que interagem com ele, além de permitir alterá-lo sem a necessidade de atualizar os outros.

Muitos *softwares* são desenvolvidos de maneira semelhante à modularização, divididos em partes, tendo cada parte uma responsabilidade. Porém não são realmente modularizados, ou seja, não atendem ao conceito de modularização como descrito acima. Dois exemplos de projetos de *softwares* que foram desenvolvidos utilizando a especificação OSGi e que atendem a definição de modularização, são, IDE¹ Eclipse, ferramenta de desenvolvimento de softwares e o GlassFish, servidor de aplicações JEE².

O uso da modularização traz grandes benefícios para o desenvolvimento e manutenção de um *software*. Poder parar parte de uma aplicação para realizar uma manutenção ou poder instalar novas funcionalidades, garantindo que todas as outras partes restantes continuem funcionando normalmente, seria uma característica notável da aplicação, principalmente em grandes empresas, que não se pode parar todo o sistema para atualizar um relatório, por

1 IDE – Abreviação para *Integrated Development Environment*.

2 JEE – Abreviação para *Java Platform, Enterprise Edition*.

exemplo.

Fernandes (2009) forneceu uma visão geral sobre OSGi, mostrando seus benefícios e salientando a importância da plataforma Java possuir um melhor suporte à modularidade, até demonstrando com um exemplo simples as premissas e vantagens do OSGi.

Mayworm (2010) demonstra a tecnologia OSGi no contexto de aplicações distribuídas, permitindo a disponibilização de seus serviços remotamente, integrando com diferentes *frameworks* de *middleware* para o desenvolvimento de aplicações empresariais.

Malcher (2008) apresenta um modelo de componentes adaptável para se usar em ambientes distribuídos. Também analisa alguns aspectos, como modelo de distribuição, transparência, descoberta de novos módulos disponíveis, desempenho e performance dos mesmos. Afirmar ainda que para se escolher entre um modelo de distribuição – *deployment* local ou execução remota – é necessário analisar o contexto e o objetivo da aplicação, pois cada um se adapta melhor a determinadas situações e ambientes de execução.

Portanto, após despertar um grande interesse pelo modelo de desenvolvimento modular, será realizado estudos sobre a especificação OSGi, e ainda, o desenvolvimento de uma aplicação modular que demonstre tal modelo de desenvolvimento.

O presente trabalho possui como objetivo geral:

- Demonstrar o modelo de desenvolvimento modular utilizando a especificação OSGi com foco em aplicações empresariais;

Os objetivos específicos são:

- Pesquisar as melhores práticas e *frameworks* que contribuem para a produtividade no desenvolvimento modularizado;
- Desenvolver uma aplicação que exemplifique o modelo de desenvolvimento modular através da especificação OSGi;
- Obter através dos resultados uma conclusão sobre as vantagens do modelo de desenvolvimento modular.

Cada vez mais as empresas necessitam automatizar o gerenciamento de suas atividades, havendo a necessidade do *software* utilizado acompanhar esse crescimento com a

inclusão de novas funcionalidades, isso é impulsionado pela necessidade de acompanhar o surgimento de novos segmentos de mercado, conveniência de enxugar o produto para derrubar barreiras de entrada, adição de novos relatórios rapidamente ou corresponder às regras do negócio, do governo ou do ramo atuante da empresa.

Diante dos vários segmentos e constantes mudanças empresariais, novos *softwares* são desenvolvidos e precisam estar preparados para acompanhar as modificações que ocorrem nos processos de uma empresa. Além de sistemas antigos já utilizados terem de ser expandidos ou modificados para acompanhar a necessidade da empresa. Dessa forma o desenvolvimento separado em módulos seria uma solução para atender empresas de diferentes setores, ao contrário de criar um *software* para cada ramo empresarial. Além disso a utilização de módulos em um sistema torna mais flexível sua manutenção, pois a mesma é realizada somente no módulo desejado, não afetando o restante da aplicação.

O trabalho tem relevância na visão acadêmica por agregar conceitos sobre o desenvolvimento de *softwares* de forma modularizada utilizando a tecnologia OSGi. E ainda, por ser pouco utilizada no mercado devido à sua complexidade de aprendizagem, este trabalho visa reunir informações e disponibilizar uma documentação com práticas e vantagens que essa tecnologia pode oferecer.

Desenvolvedores de *softwares* procuram sempre a utilização de novas práticas e tecnologias produtivas. O modelo de desenvolvimento modular utilizando a tecnologia OSGi oferece diversos benefícios que ajudam na expansão e manutenção de uma aplicação. Isto também se torna útil para as empresas, pois para novas funcionalidades seriam criados módulos, e a manutenção do sistema seria realizada somente no módulo específico, sem a necessidade de parar o restante do sistema.

Os módulos encontrados no processo de modularização, podem se tornar componentes. Esta componentização, por sua vez, ocorre quando um ou mais módulos se tornam uma parte física e substituível de um sistema e com o qual o respectivo componente está em conformidade e disponibilizado através de um conjunto de interfaces. Isso resulta em reutilização de código, sistemas otimizados e pouca redundância (BOOCH, 2000; BEZERRA, 2002).

2 QUADRO TEÓRICO

Para que se possa desenvolver qualquer solução, é necessário o uso de algumas ferramentas, teorias e tecnologias. Neste capítulo é apresentada algumas delas, que serão utilizadas no desenvolvimento deste trabalho, bem como sua utilidade.

2.1 Modularização

O conceito de modularização surgiu como uma solução aos antigos *softwares* monolíticos, pois representavam grandes dificuldades de entendimento e manutenção. Esse conceito, afirma que a complexidade do *software* pode ser dividida em partes menores, resolvendo assim problemas separadamente (USP, 2015).

A modularidade é um fator interno no desenvolvimento de *softwares*, que influencia significativamente em fatores externos, como: qualidade, extensibilidade e reusabilidade do mesmo. O desenvolvimento de um sistema modular é realizado através da composição de partes pequenas para formar partes maiores. Essas partes são chamadas de módulos (BORBA, 2015).

Segundo Knoernschild (2012), para que uma aplicação seja modularizada, seus módulos devem ser instaláveis, gerenciáveis (parar, reiniciar e desinstalar sem interromper o restante da aplicação), reutilizáveis, combináveis, não guardar estado e oferecer uma interface clara.

A granularidade de um *software* é definida a partir de como o sistema é dividido em partes. Sendo assim partes maiores tendem a fazer mais operações do que partes menores. Isso influencia diretamente no uso e reúso de código, como demonstra a Figura 1.

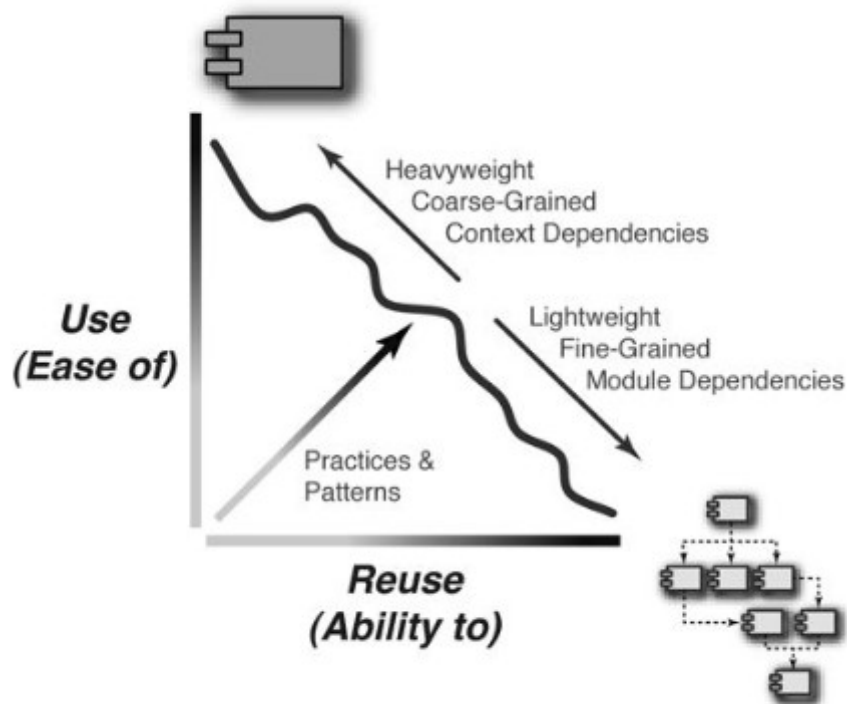


Figura 1 – Uso e reúso de partes do *software*. **Fonte:** Java Application Architecture (2012).

Conforme mostra a Figura 1, quanto maior a granularidade dos módulos do sistema, torna-se mais fácil usá-los e mais difícil de reutilizá-los. Do contrário, quanto menor a granularidade, torna-se mais fácil reutilizá-los, e mais difícil usá-los (KNOERNSCHILD, 2012).

Muitos sistemas são desenvolvidos utilizando o modelo de camadas. Neste modelo, também existe a granularidade, pois quanto mais se caminha para baixo da hierarquia de camadas, menor é a granularidade, enquanto as camadas superiores da hierarquia possuem uma granularidade maior. Esse *design* de camadas é demonstrado na Figura 2.

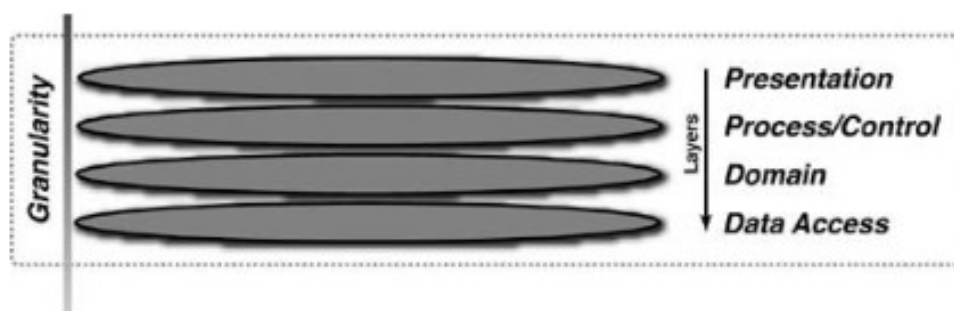


Figura 2 – Arquitetura de camadas. **Fonte:** Java Application Architecture (2012).

Knoernschild (2012) diz que, diferente da arquitetura de camadas mostrada anteriormente, a arquitetura de sistemas modulares em Java, através da tecnologia OSGi, propõe um desenvolvimento modularizado conforme mostra a Figura 3.

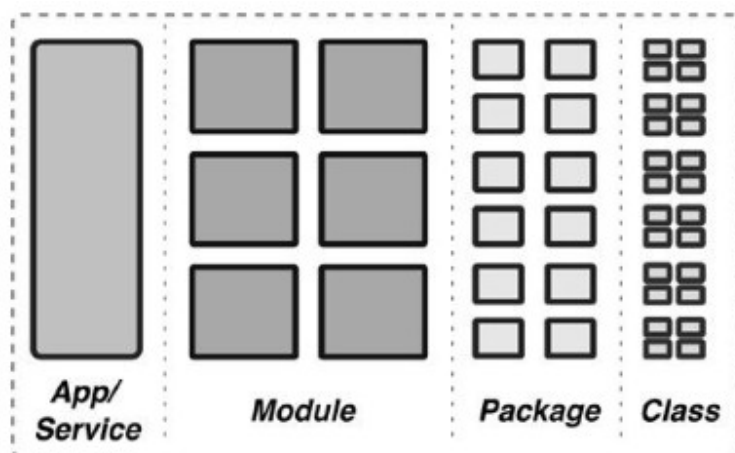


Figura 3 – Arquitetura modular OSGi. **Fonte:** Java Application Architecture (2012).

Nesse modelo, o desenvolvedor cria os módulos com classes e pacotes que são disponibilizados para o restante do sistema através de interfaces que se tornam serviços para outros módulos.

Um *software* modular reduz a complexidade, facilita a mudança e resulta em uma implementação mais fácil ao estimular o desenvolvimento paralelo das partes do sistema, desta forma aumentando a produtividade (WERLANG; OLIVEIRA, 2006).

Werlang e Oliveira (2006) afirmam que módulos independentes são mais fáceis de desenvolver, manter e testar, pois os efeitos secundários provocados por modificações são limitados.

Knoernschild (2012) reforça isso, afirmando que, uma das vantagens da modularidade, é o impacto reduzido que as alterações num *software* modularizado podem causar. A Figura 4, demonstra a representação do impacto em aplicações modulares e não modulares quando é realizada alguma alteração no sistema.

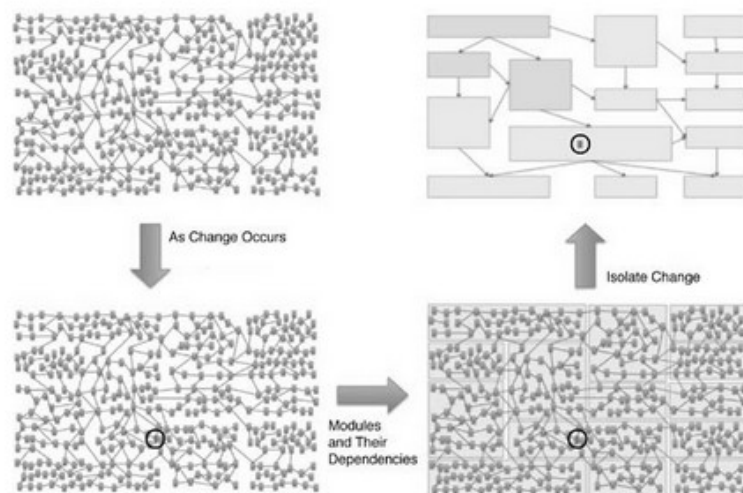


Figura 4 – Impacto das mudanças. **Fonte:** Java Application Architecture (2012).

Em uma aplicação não modularizada, qualquer alteração pode gerar um impacto na aplicação toda. Já no sistema modular as alterações geram impacto somente dentro do módulo, já que os mesmos oferecem interfaces bem definidas e suas operações estão encapsuladas, ou seja, não são conhecidas por módulos externos.

O quanto os módulos são independentes é determinado levando em consideração dois critérios: coesão e acoplamento. A coesão é a relação existente entre as responsabilidades e as unidades dos módulos. Onde um módulo coeso tem responsabilidades e propósitos claros e bem definidos. O acoplamento é o grau em que um módulo depende de outros, com isso o acoplamento surge em função do relacionamento existente entre os módulos e é caracterizado pelo uso de serviços entre eles (STAA, 2000).

Pressman (1995) estabelece que um módulo coesivo executa uma única tarefa dentro do *software*, enquanto o nível de acoplamento deve ser o mais baixo possível, utilizando um número mínimo de informações trocados entre os módulos.

Na modularização deve-se buscar a alta coesão com independência funcional para permitir que futuras modificações não comprometessem o funcionamento do sistema. A simples conexão entre os serviços dos módulos resulta em um *software* que é mais fácil de entender e menos propenso a propagação de erros pelo sistema (PRESSMAN, 1995).

2.2 Especificação OSGi

Grandes aplicações enfrentam um problema conhecido no seu desenvolvimento, a complexidade. Uma maneira de resolver esse problema é quebrar o sistema em partes menores, ou módulos. Devido a sua flexibilidade, a linguagem de programação Java permitiu construir sobre sua plataforma a tecnologia OSGi, que oferece suporte à construção de sistemas modulares (FERNANDES, 2009).

A especificação OSGi é mantida pela OSGi Alliance, um consórcio mundial formado por empresas tecnológicas, essa associação cria especificações que permitem o desenvolvimento modular através da tecnologia Java. OSGi é um conjunto de especificações, que segue um modelo de desenvolvimento em que as aplicações são dinamicamente formadas por componentes distintos e reutilizáveis (OSGI ALLIANCE, 2015).

A especificação teve início em março de 1999 pela própria OSGi Alliance. Seus principais desafios na época não era desenvolver uma solução para a execução de diferentes versões de um mesmo projeto na mesma aplicação, mas sim de elaborar uma maneira de diferentes componentes que não se conhecem possam ser agrupados dinamicamente sob o mesmo projeto (OSGI ALLIANCE, 2015).

De acordo com OSGi Alliance (2015), essa tecnologia possui como benefício a redução da complexidade do desenvolvimento de modo em geral, aumento da reutilização de módulos, incremento da facilidade de codificação e teste, gerenciamento total dos módulos sem a necessidade de reiniciar a aplicação, aumento do gerenciamento da implantação e detecção antecipada de *bugs*.

Os *bundles* – como os módulos são chamados no contexto da OSGi – consomem e/ou disponibilizam serviços. Eles estão organizados de tal maneira que formam a tríade consumidor, fornecedor e registro de serviços, na qual pode-se consumir serviços ou disponibilizá-los. Para ativar um novo serviço, o mesmo deve ser registrado como tal no *framework*, possuindo visibilidade por parte de *bundles* externos (OSGI ALLIANCE, 2015).

A OSGi Alliance (2015), define que o *framework* OSGi utiliza uma arquitetura de camadas em sua implementação, conforme demonstrado na Figura 5.

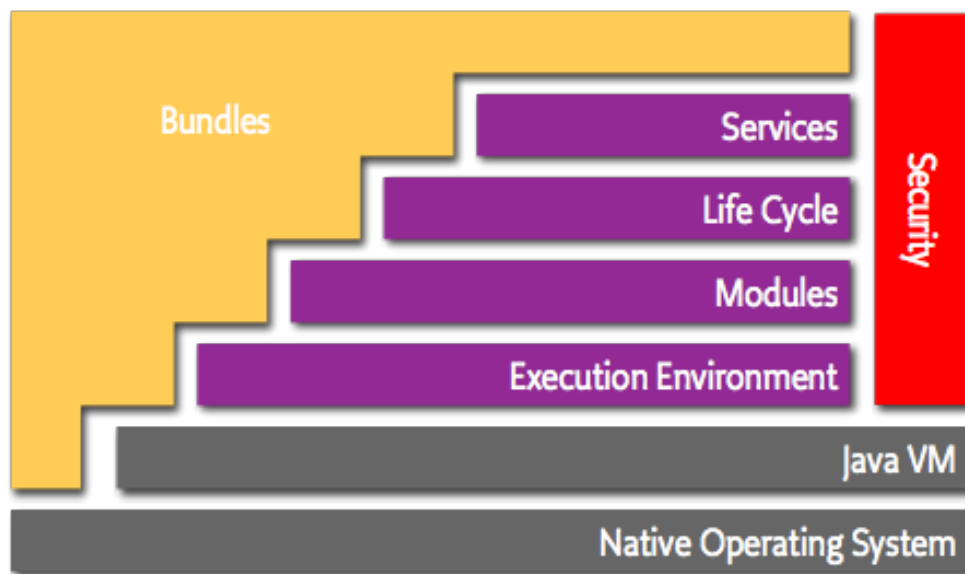


Figura 5 – Estrutura de camadas. Fonte: OSGi Alliance (2015).

As camadas do *framework* OSGi podem ser assim explicadas:

- **Bundles:** são os componentes (módulos) criados pelos desenvolvedores de *softwares*;
- **Services:** camada responsável por conectar os *bundles* de forma dinâmica oferecendo um modelo *publish-find-bind*³. Esta camada permite registrar e obter serviços dos módulos;
- **Life Cycle:** camada que provê uma API⁴ para instalar, iniciar, parar, atualizar, e desinstalar os *bundles*;
- **Modules:** camada que administra a importação e exportação de códigos dos *bundles*.
- **Execution Environment:** camada em que é definida qual implementação do OSGi está sendo utilizada.
- **Security:** camada responsável por gerenciar aspectos de segurança do *framework*.

A especificação OSGi define uma camada chamada *Life Cycle* que gerencia o ciclo de vida dos *bundles* e provê uma API que permite o desenvolvedor instalar, desinstalar, iniciar, parar e atualizar os *bundles* (BOSSCHAERT, 2012).

Bartlett (2009) afirma que um *bundle* pode passar por vários estados em seu ciclo de vida. Na Figura 6, é demonstrado cada um desses estados e suas funções.

3 *Publish-find-bind*: publicar, encontrar e vincular (tradução nossa).

4 API – Abreviação para *Application Programming Interface*.

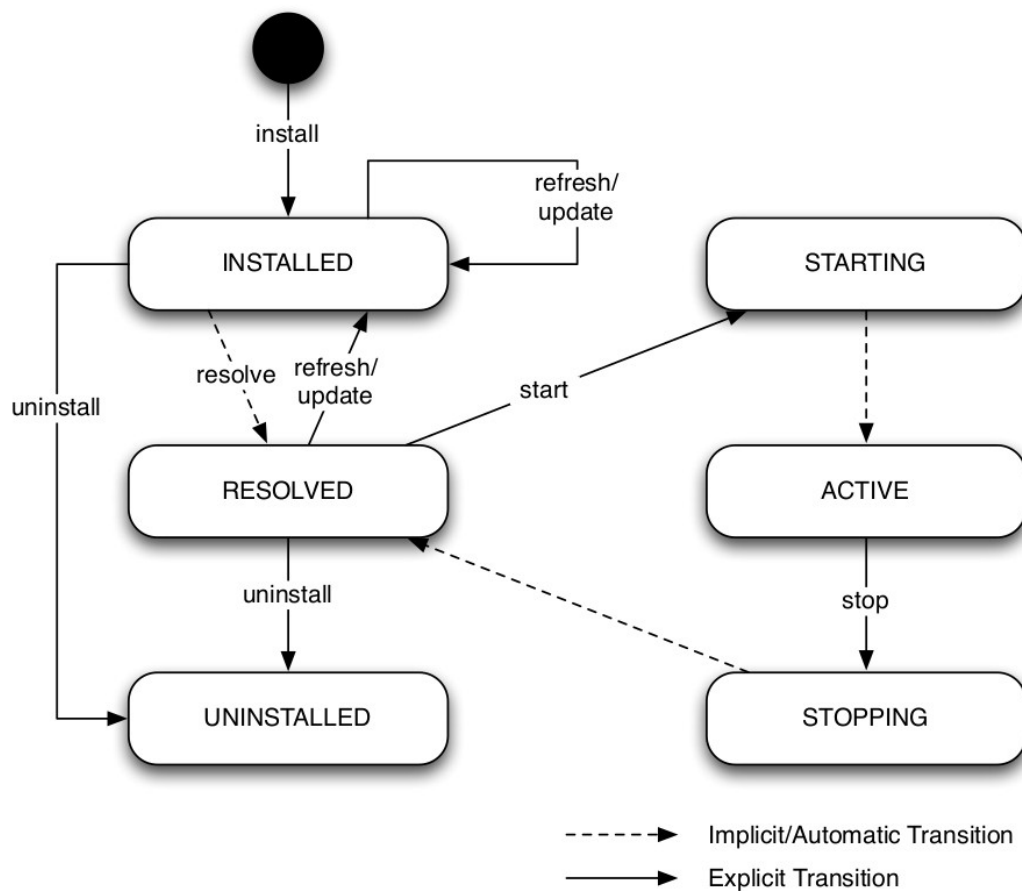


Figura 6 – Estados dos *bundles*. Fonte: OSGi In Practice (2009).

Esses estados e funções podem ser assim explicados:

- **Installed:** o *bundle* é instalado com êxito;
- **Resolved:** avalia se o *bundle* está pronto para ser iniciado ou parado, validando as informações dos *bundles*, como seus pacotes e versões, e ainda a versão do Java necessária para executar o *bundle*;
- **Starting:** estado de transição, neste estado o método `BundleActivator.start()` do *bundle* é invocado para que o mesmo seja iniciado. Por ser um estado de transição, nenhum *bundle* fica parado permanentemente nesse estado, transitando automaticamente para outro;
- **Active:** nesse estado o *bundle* está validado e em funcionamento, consumindo e/ou disponibilizando serviços, somente esperando alguma requisição para ser utilizado;

- **Stopping:** também é um estado de transição, onde o método `BundleActivator.stop()` é chamado para parar o funcionamento do *bundle*, resultando na transferência para o estado *Resolved*;
- **Uninstalled:** estado em que o *bundle* se encontra desinstalado e o mesmo não é mais representado no *framework*. Se o *bundle* for reinstalado, ele assume o papel de um novo *bundle*.

Segundo Fernandes (2009), após a instalação do *bundle*, o mesmo fica armazenado em um meio persistente do *framework* até ser desinstalado explicitamente.

Um aspecto importante é que um *bundle* pode ser executado em qualquer implementação OSGi. Ou seja, um *bundle* desenvolvido e testado em uma implementação pode ser executado em qualquer outra implementação. (LUCENA, 2010).

O *framework* é o centro da especificação da plataforma OSGi, definindo um modelo para o gerenciamento do ciclo de vida da aplicação, registro dos serviços e ambiente de execução (FERNANDES, 2009).

Segundo Bartlett (2009), a OSGi Alliance apenas define uma especificação do *framework*, existindo várias implementações nos dias atuais, mas as quatro a seguir merecem destaque por serem as mais conhecidas e utilizadas.

- **Equinox:** é o *framework* mais usado atualmente, sua grande popularidade provém de fazer parte do gerenciador de *plug-ins* do Eclipse. Pode ser encontrado em muitos outros *softwares*, como IBM Lotus Notes e Websphere Application Server. Encontra-se sob a licença Eclipse Public License (EPL) e implementa a versão 4.1 das especificações OSGi.
- **Knopflerfish:** é uma implementação bem madura e popular da versão 4.1 da especificação. É desenvolvido e mantido pela empresa Sueca Makewave AB, a qual oferece seu produto tanto em uma versão gratuita sob a licença BSD⁵ quanto uma versão comercial com suporte oferecido pela empresa.
- **Felix:** é a implementação mantida pela Apache Software Foundation. Implementa a versão 5 da especificação OSGi e possui foco na compactação e facilidade de incorporação do *bundle* na aplicação. Está sob a licença Apache 2.0.
- **Concierge:** é uma implementação bem compacta e altamente otimizada da

5 BSD – Abreviação para *Berkeley Software Distribution*.

especificação versão 3. É mais utilizado para plataformas com recursos limitados, como aplicações móveis e embarcadas. Se encontra sob a licença BSD.

A única diferença entre um *bundle* e um arquivo tradicional é uma pequena quantidade de metadados adicionado ao arquivo MANIFEST.MF. Com isso, caso deseje-se usar um *bundle* JAR fora de um contêiner OSGi, não haverá nenhum problema (FERNANDES, 2009).

De acordo com Vogel (2015) os principais metadados e suas respectivas características são:

- **Bundle-Name:** nome ou breve descrição do *bundle*;
- **Bundle-SymbolicName:** o identificador único do *bundle*;
- **Bundle-Version:** define a versão do *bundle* e deve ser incrementado sempre que uma nova versão é publicada;
- **Bundle-Activator:** Define uma classe opcional que implementa a interface BundleActivator.java. Uma instância dessa classe é criada quando o *bundle* é ativado e seus métodos **start()** e **stop()** são chamados sempre que o módulo é iniciado ou parado. Essa classe pode ser utilizada para configurar o *bundle* durante a inicialização ou realizar alguma operação antes dele ser parado;
- **Bundle-RequiredExecutionEnvironment:** especifica qual versão Java é necessária para executar o *bundle*. Se esta exigência não for cumprida, o *framework* não inicializa o *bundle*;
- **Bundle-ActivationPolicy:** configuração que permite definir se o *framework* irá inicializar os *bundles* de forma *Lazy* ou não, se ativar essa opção o *bundle* só será ativado caso alguma classe, interface ou serviço necessite do mesmo;
- **Bundle-ClassPath:** especifica o diretório de onde será carregado as classes do *bundle*. O padrão é ' ' pois permite que classes possam ser carregadas a partir da raiz do pacote;
- **Web-ContextPath:** esse atributo é somente utilizado para projetos do tipo WAR. Seu valor corresponde à URL definida para a aplicação *web*;
- **Export-EJB:** esse atributo é somente utilizado para projetos do tipo EJB. O mesmo define quais classes com anotações da especificação EJB serão disponibilizadas via serviços do OSGi. Quando se utiliza o valor “ALL” todas as classes com anotações

são disponibilizadas como serviços. Usa-se “NONE” para não disponibilizar nenhuma classe como serviço, e havendo a necessidade de disponibilizar apenas alguns serviços, coloca-se o nome da classe separado por vírgulas;

- **Import-package:** lista dos pacotes requeridos pelo módulo para o seu funcionamento com a sua devida versão especificada.
- **Export-package:** lista dos pacotes visíveis e utilizáveis pelos módulos externos.

O OSGi especifica um conceito de versionamento de componentes, em que os módulos podem trabalhar com versões diferentes de um mesmo serviço utilizado. Seguindo uma consistente estrutura de numeração, utilizando três segmentos de números e um segmento alfanumérico, são eles, *major*, *minor*, *micro* e *qualifier*, respectivamente. Exemplo, "1.2.3.beta_1". Esses segmentos também podem ser omitidos, formando por exemplo a versão "1.2". Realizar esse versionamento se torna necessário para resolver o problema de incompatibilidade de versões entre módulos (FERNANDES, 2009).

A tecnologia OSGi demonstra uma arquitetura modular ágil, incluindo um mecanismo de ciclo de vida dinâmico, que permite a criação de produtos e serviços em diversos setores, como demonstra a Figura 7. Também assegura a gestão remota e interoperabilidade de aplicações e serviços em uma ampla variedade de dispositivos, isso devido à facilidade na componentização e interação entre *softwares* modulares (OSGI ALLIANCE, 2015).

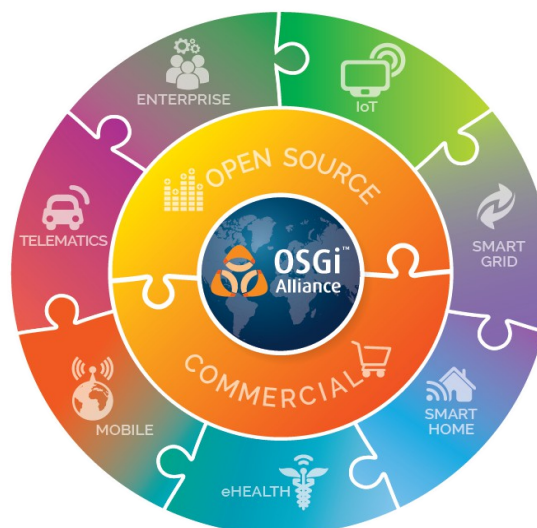


Figura 7 – OSGi atuando em diferentes setores. **Fonte:** OSGi Alliance (2015).

De acordo com Fernandes (2009), os *Working Groups*, criados pelo conselho administrativo da OSGi Alliance, são subdivisões responsáveis pelo desenvolvimento técnico de cada área de atuação. Estão entre os *Working Groups* a *Core Platform Expert Group* (CPEG), *Mobile Expert Group* (MEG), *Vehicle Expert Group* (VEG), *Enterprise Expert Group* (EEG) e *Residential Expert Group* (REG). Dessa forma, à medida que é necessário explorar e desenvolver algo novo, são criadas novas *Working Groups*, para analisar seus requisitos de mercado e especificar soluções que a atendam, como é o caso da nova *Working Group* da subdivisão de IoT⁶.

2.3 Java

A linguagem de programação Java é composta por coleções de classes existentes nas bibliotecas de classe Java, conhecidas como Java APIs. É possível criar classes para formar uma aplicação Java e também utilizar as coleções de classes da Java API. Portanto, para se desenvolver uma aplicação utilizando essa linguagem, é preciso entender como criar classes próprias que irão compor a aplicação e como trabalhar utilizando as classes da Java API (DEITEL, 2010).

Java foi lançada em 1996 pela Sun Microsystems⁷ com a finalidade de desenvolver aplicativos para diferentes plataformas (Windows – Linux – *Web* – *Mobile*) sem a necessidade de alterar o código entre elas. Qualquer dispositivo que possua a JVM⁸ é capaz de executar um *software* desenvolvido em Java (CLARO e SOBRAL, 2008).

Java oferece uma plataforma para o desenvolvimento de sistemas de médio a grande porte. Uma das muitas vantagens de utilizá-la é a grande quantidade de bibliotecas gratuitas que podem ser utilizadas em diversos tipos de projetos. Como cada linguagem é apropriada para uma determinada área, a utilização de Java é melhor para o desenvolvimento de aplicações que tenham tendência a crescer (CAELUM, 2015c).

Para desenvolver aplicações em Java é necessário instalar um Kit de desenvolvimento, o Java *Development Kit* – JDK, o qual pode ser obtido no próprio site da Oracle – empresa mantenedora da plataforma. Ele é composto de compilador, máquina virtual, bibliotecas e

6 IoT – Abreviação para *Internet of Things*.

7 Fabricante de computadores, semicondutores e softwares adquirida pela Oracle Corporation em 2009.

8 JVM – Abreviação para Java Virtual Machine.

utilitários (CAELUM, 2015c).

O Java EE, abreviação para Java *Enterprise Edition*, consiste de uma série de especificações bem detalhadas, fornecendo toda uma infraestrutura para o desenvolvedor utilizar. Os serviços dessa infraestrutura compreendem *web services*, gerenciamento de *threads*, gerenciamento de conexões HTTP, gerenciamento de sessão *web*, etc. A Sun Microsystems criou essa especificação para diminuir os custos dos projetos que utilizam Java, pois utilizando Java EE não teria a necessidade de se desenvolver novamente as funcionalidades citadas anteriormente. Porém Java EE é apenas um conjunto de especificações, sendo necessário utilizar uma implementação dessas especificações (CAELUM, 2015a).

Uma aplicação Java *Web* gera páginas *web* interativas, que contêm vários tipos de linguagem de marcação (HTML, XML, etc.) e conteúdo dinâmico. Normalmente é composto por componentes *web*, *servlets* e *JavaBeans* para modificar e armazenar dados temporariamente, interagir com bancos de dados e *web services* e processar o conteúdo como resposta às solicitações do cliente (NETBEANS, 2015b). Através dessa tecnologia é possível ter uma grande economia de tempo e recursos de desenvolvimento, uma vez que a aplicação será executada na aplicação cliente universal, o navegador.

EJB que significa *Enterprise JavaBeans*, é a arquitetura de componentes do lado servidor para a plataforma Java EE. Nela são encapsulados as lógicas de negócio de uma aplicação, ou seja, os objetivos da aplicação são implementados nessa parte do projeto. Utilizá-la permite o desenvolvimento rápido e simplificado de aplicativos transacionais, distribuídos, seguros e portáteis baseados na tecnologia Java (ORACLE, 2015).

A especificação JSR-299 referente ao CDI, abreviatura para *Context Dependency Injection*, é uma API para injeção de dependências e contextos sendo parte integrante do Java EE e fornece uma arquitetura que permite aos seus componentes, como *servlets*, *enterprise beans* e *JavaBeans* existirem dentro do ciclo de vida de uma aplicação com escopos bem definidos. A injeção de dependências é executada através do uso de anotações, dentre elas a mais conhecidas são `@Inject`, `@RequestScoped`, `@Singleton`, `@ApplicationScoped` (NETBEANS, 2015a).

Toda a plataforma Java demonstra ser de imprescindível utilidade neste trabalho, pois além das vantagens citadas anteriormente, a especificação OSGi é desenvolvida utilizando a mesma.

2.4 Apache Felix

Apache Felix ou Felix é uma implementação da especificação OSGi, que implementa sua versão 5. Atualmente é mantida pela Apache Software Foundation, sob licença Apache 2.0 (FERNANDES, 2009).

Apache (2015a) afirma que, o *framework* Felix permite que os códigos desenvolvidos possam funcionar em qualquer outro *container* OSGi, permitindo a troca de implementações sem problemas de compatibilidade. Sua comunidade está sempre se esforçando para que não haja esse problema.

Utilizar uma implementação de fácil utilização no desenvolvimento de uma aplicação, que oferece desde a mais simples base de *software* até a possibilidade de integração total com outros *softwares*, como os servidores de aplicação GlassFish e Jonas, nos auxilia no desenvolvimento do projeto, pois permite, além das facilidades para implementar a aplicação, a opção de migração para outra implementação OSGi (BARTLETT, 2009).

2.5 GlassFish

Em 2005, a Sun Microsystems criou o projeto GlassFish com o objetivo principal de produzir um servidor de aplicações JEE. Atualmente se encontra na versão 4.1, e é mantido pela Oracle, que adquiriu a Sun em 2010. (GONÇALVES, 2010).

Segundo DevMedia (2011), o servidor de aplicações GlassFish possui uma versão paga e outra de código *Open Source*. A versão livre fornece um servidor de aplicações com o código fonte disponibilizado para *download* no site do GlassFish. Possui integração com diversas IDEs, como NetBeans, Eclipse ou IntelliJ.

O GlassFish é um servidor de aplicação de fácil utilização, além de ser leve e modular. Também fornece armazenamento em *cluster* com alta disponibilidade, console de administração baseado na web, e REST APIs (ORACLE, 2011).

Devido às vantagens e à grande integração com o *framework* Felix, a utilização do GlassFish será útil no desenvolvimento de uma aplicação modular *web*.

2.6 RESTful Web Services

Em 2000, o cientista Roy Fielding, apresentou em sua tese de doutorado uma nova forma de integrar sistemas distribuídos, esse novo modo se chamava REST, que significa *Representational State Transfer* (SANTOS, 2009).

De acordo com Santos (2009), REST é um estilo de arquitetura de *software* para sistemas hipermídia distribuídos, onde utilizamos um navegador web para acessar recursos, mediante a digitação de uma URL⁹.

Foi concebido baseando-se em HTTP¹⁰ e devido a isso sua arquitetura é baseada em cliente-servidor e seus serviços não possuem estado (*stateless*). Empregar o uso do protocolo HTTP e de URIs¹¹ torna as aplicações mais simples, leves e com alta performance (SANTOS, 2009).

Segundo Santos (2009), os recursos dos serviços são obtidos de um servidor *web* após enviar a requisição, podendo o endereço dessa requisição ser denominado:

- URL representa o caminho percorrido, separando em diretórios.
- URI são os identificadores de acesso a recursos específicos que utilizando junto à URL aponta para o recurso específico.

Uma vez que o recurso é identificado, é necessário definir a ação que irá se tomar utilizando este recurso, em REST estas operações são GET, PUT, POST e DELETE. Essas operações são mapeadas com as operações CRUD (*Create, Retrieve, Update, Delete*) de um banco de dados.

Conforme Saudate (2013), os recursos e operações são obtidos e salvos através de um tipo de dado, este pode ser XML¹² ou JSON¹³, cada um com suas vantagens e características próprias, onde:

- XML é uma linguagem de marcação extensível. Desta forma, conseguimos expressar grande parte das informações utilizando este formato. Entre as características

9 URL – Abreviação para *Uniform Resource Locator*.

10 HTTP – Abreviação para *HyperText Transfer Protocol*.

11 URI – Abreviação para *Uniform Resource Identifier*.

12 XML – Abreviação para *eXtensible Markup Language*.

13 JSON – Abreviação para *JavaScript Object Notation*.

principais dessa linguagem de marcação temos:

- Tem um e apenas um elemento raiz, porém este é flexível o bastante para ter qualquer nome;
 - Possuem seções específicas para fornecimento de instruções de processamento, prólogo e epílogo, onde estas se localizam antes ou após os dados, respectivamente;
 - As estruturas mais básicas em um XML são as *tags* e os atributos, com isso os dados transportados pela sua aplicação podem estar presentes tanto na forma de atributos como de conteúdo das *tags*.
- JSON é uma linguagem de marcação criada por Douglas Crockford. Tem por principal vantagem o tamanho reduzido em relação ao XML, e devido a isso acaba sendo usado quando o cenário apresenta o consumo de banda como um recurso crítico. Suas principais características são:
 - Um objeto contém zero ou mais membros;
 - Um membro contém zero ou mais pares e zero ou mais membros;
 - Um par contém uma chave e um valor;
 - Um membro também pode ser um *array*.

De acordo com Jersey (2015), a especificação JAX-RS define e regulamenta as regras do JSR 311, porém para se utilizar das vantagens oferecidas, é necessário usar uma implementação. Dessa maneira será utilizado o *framework opensource* Jersey.

Utilizar Jersey se torna de grande vantagem pois ele é muito mais do que a implementação de referência da especificação, fornecendo sua API que estende o JAX-RS, e utilitários adicionais, simplificando o serviço RESTful e o desenvolvimento do cliente da aplicação (JERSEY, 2015).

2.7 Maven

O Maven é um projeto de código livre, mantido pela Apache Software Foundation, foi criado para gerenciar o processo de criação do projeto Jakarta Turbine. Com o lançamento das novas versões, ele passou de uma simples ferramenta de construção para uma ferramenta complexa de gestão de construção de *software* (SANTOS, 2008).

Segundo Apache (2015b), o Maven possui como principal finalidade o entendimento do estado de desenvolvimento de um *software* em um curto espaço de tempo. Para alcançar este objetivo, é necessário atingir alguns outros, como:

- Facilitar o processo de desenvolvimento;
- Prover um sistema de desenvolvimento uniforme;
- Disponibilizar informações importantes sobre o projeto;
- Prover orientação para atingir melhores práticas de desenvolvimento.

Santos (2008) apresenta um conjunto de modelos conceituais que explicam o funcionamento desta ferramenta:

- **Project object model (POM):** é o componente principal de configurações para o funcionamento, onde o desenvolvedor informa em um arquivo pom.xml as dependências de que necessita em seu projeto, além de outras configurações;
- **Dependency management model:** a gestão de dependência é uma importante fase do processo de construção do *software*, quando se utiliza projetos e dependências de terceiros, é comum ter muito trabalho e problemas na obtenção e gerenciamento das dependências de nossas dependências. Nesse quesito, o Maven é uma das melhores ferramentas para a construção de projetos complexos, pela sua robustez na gestão dessas dependências;
- **Ciclo de vida de construção e fases:** associado ao POM, existe a noção de ciclo de vida de construção e fases. Onde é criada uma conexão entre os modelos conceituais e os modelos físicos;
- **Plug-ins:** estendem as funcionalidades do Maven.

No desenvolvimento modular é necessário que seja realizado as dependências existentes entre cada módulo do sistema. Controlar tais dependências manualmente demandará um grande esforço e tempo, dessa forma, utilizar o Maven para gerenciar a obter as dependências dos módulos, se torna de suma importância, pois auxilia na redução do tempo de desenvolvimento, implantação e manutenção do *software*.

2.8 PostgreSQL

Segundo Neto (2003 apud Souza; Amaral; Lizardo, 2012, p. 2), o PostgreSQL é um Sistema Gerenciador de Banco de Dados Relacional (SGBDR) que está baseado nos padrões SQL¹⁴ ANSI-92, 96 e 99. Possui alta performance, fácil administração e utilização em projetos pelos *Database Administrators* (DBAs) e desenvolvedores de *softwares*.

O PostgreSQL teve origem em um projeto chamado de POSTGRES na Universidade de Berkeley, na Califórnia (EUA), em 1986. Sua equipe fundadora foi orientada pelo professor Michael Stonebraker com o apoio de diversos órgãos, dentre eles o Army Research Office (ARO) e o National Science Foundation (NSF). Atualmente, o SGBD encontra-se em sua versão 9.4 estável, contendo todas as principais características que um SGBD pode disponibilizar (MILANI, 2008).

Apesar da pouca popularidade, o PostgreSQL é um gerenciador de banco de dados veloz, robusto e se encontra na lista dos que mais possuem recursos, além de ser o pioneiro na introdução de vários conceitos objeto relacionais (STERN, 2003).

Seu código é livre e há um grupo responsável pela sua validação. Grandes empresas como a Fujitsu, NTT¹⁵ Group, Skype, Hub.org, Red Hat e SRA¹⁶ são financiadoras do PostgreSQL que, além disso, recebe doações. Ele é utilizado por multinacionais, órgãos governamentais e universidades. Recebeu vários prêmios como melhor sistema de banco de dados *Open Source* (SOUZA; AMARAL; LIZARDO, 2012).

14 SQL – Abreviação para *Structured Query Language*.

15 NTT – Abreviação para *Nippon Telegraph and Telephone*.

16 SRA – Abreviação para *Systems Research and Applications Corporation*.

2.9 HyperText Markup Language (HTML)

HyperText Markup Language, é o significado da sigla HTML, que, em português, significa linguagem para marcação de hipertexto. Ela foi criada por Tim Berners-Lee na década de noventa tornando-se um padrão internacional. De modo geral, o hipertexto é todo o conteúdo de um documento para *web*, com a característica de se interligar a outros documentos da *web* através de links presentes nele próprio (SILVA, 2011).

Conforme Mozilla Developer Network (2014), a HTML é uma linguagem de marcação que estrutura o conteúdo de um documento *web*. O conteúdo visto ao acessar uma página através do navegador é estruturado e descrito utilizando a HTML, tornando-se a principal linguagem para conteúdo *web* mantida pelo *World Wide Web Consortium* ou resumidamente W3C (W3C, 2015).

O W3C é uma comunidade internacional liderada pelo criador da *web* Tim Berners-Lee, é formada por organizações, profissionais e público em geral com o objetivo de conduzir a *web* ao seu potencial máximo, através do desenvolvimento de padrões e especificações (W3C, 2015).

Mesmo sendo uma linguagem destinada à criação de documentos, a HTML não tem como objetivo definir estilos de formatação, como por exemplo, nomes e tamanhos de fontes, margens, espaçamentos e efeitos visuais. Também não possibilita adicionar funcionalidades de interatividade avançada à página. A linguagem HTML destina-se somente a definir a estrutura dos documentos *web*, fundamentando dessa maneira os princípios básicos do desenvolvimento seguindo os padrões *web* (SILVA, 2011).

2.10 JavaScript

JavaScript é uma linguagem de programação criada pela Netscape em parceria com a Sun Microsystems. Sua primeira versão, definida como JavaScript 1.0, foi lançada em 1995 e implementada em março de 1996 no navegador Netscape Navigator 2.0 (SILVA, 2010).

Segundo Silva (2010), o JavaScript tem como finalidade fornecer funcionalidades para

adicionar interatividades avançadas a uma página web. É desenvolvido para ser executado no lado do cliente, ou seja, é interpretado pelo navegador do usuário. Os navegadores possuem funcionalidades integradas para realizar a interpretação e o funcionamento da linguagem JavaScript.

Conforme Mozilla Developer Network (2015), JavaScript é baseado na linguagem de programação ECMAScript, o qual é padronizado pela Ecma International na especificação ECMA-262 e ECMA-402.

Entre suas características está ser uma linguagem leve para a execução, interpretada, assíncrona e baseada em objetos com funções de primeira classe. Funções de primeira classe são funções que podem ser passadas como argumentos, retornadas de outras funções, atribuídas a variáveis ou armazenadas em estrutura de dados. Além de ser executado nos navegadores, o JavaScript é utilizado em outros ambientes, como por exemplo, node.js ou Apache CouchDB (MOZILLA DEVELOPER NETWORK, 2015).

De acordo com Caelum (2015b), o JavaScript é responsável por aplicar qualquer tipo de funcionalidade dinâmica em páginas *web*. É uma linguagem poderosa, que possibilita ótimos resultados. Ferramentas como o Gmail, Google Maps e Google Docs são exemplos de aplicações *web* desenvolvidas utilizando JavaScript.

2.11 Cascading Style Sheet (CSS)

Cascading Style Sheet, é o significado da sigla CSS, que, traduzido para o português, significa folhas de estilo em cascata. Tem a finalidade de definir estilos de apresentação para um documento HTML (SILVA, 2012).

De acordo com a W3C (2015), “*Cascading Style Sheets (CSS) is a simple mechanism for adding style (e.g., fonts, colors, spacing) to Web documents*”¹⁷.

Tim Berners-Lee considerava que os navegadores eram responsáveis pela estilização de uma página web, até que em setembro de 1994, surge como proposta a implementação do CSS. Em dezembro de 1996, foi lançada oficialmente pela W3C as CSS1. O W3C utiliza o termo nível em vez de versão, tendo dessa maneira CSS nível 1, CSS nível 2, CSS nível 2.1 e

¹⁷ *Cascading Style Sheet*: folha de estilo em cascata (CSS) é um mecanismo simples para adicionar estilos (por exemplo: fontes, cores, espaçamentos) para documentos web (tradução nossa).

atualmente CSS nível 3, conhecida também como CSS3 (SILVA, 2012).

De acordo com Mozilla Developer Network (2015), a CSS descreve a apresentação de documentos HTML, XML ou XHTML¹⁸. Ela é padronizada pelas especificações da W3C e já contém seus primeiros rascunhos do nível CSS4.

Enquanto o HTML é uma linguagem destinada para estruturar e marcar o conteúdo de documentos na web, o CSS fica responsável por definir cores, fontes, espaçamentos e outros atributos relacionados a apresentação visual da página usando a marcação fornecida pelo HTML como fundamento para aplicação da estilização. Essa separação da marcação e estrutura de um documento do seu estilo de apresentação torna o uso do CSS uma grande vantagem no desenvolvimento de documentos para a web (MAUJOR, 2015).

2.12 AngularJS

AngularJS é um *framework* *opensource* mantido pela Google. Seu objetivo é estruturar o desenvolvimento *front-end* utilizando o modelo de arquitetura *model-view-controller* (MVC). O *framework* associa os elementos do documento HTML com objetos JavaScript, facilitando a manipulação dos mesmos (OLIVEIRA, 2013).

De acordo com Schimtz e Lira (2014), AngularJS é um *framework* muito poderoso que possui particularidades e funcionalidades que tornam o desenvolvimento *web* mais fácil e produtivo. Funciona como uma extensão para o documento HTML, que permite adicionar parâmetros e interação de forma dinâmica aos seus elementos, com isso adicionamos funcionalidades extras no mesmo.

2.13 Bootstrap

Bootstrap é um *framework* criado pelo Twitter que oferece uma base de estilos pronta para a criação de páginas web. É uma ferramenta *opensource*, com um nível alto de maturidade e importância no mercado (MARIE, 2015).

18 XHTML – Abreviação para *Extensible HyperText Markup Language*.

Segundo Costa (2014), o Bootstrap facilita o desenvolvimento de sites responsivos, ou seja, com a utilização deste *framework* a página web pode ser aberta em dispositivos de diferentes tamanhos sem afetar o estilo do seu conteúdo. Possui também vários componentes que podem ser utilizados sem muito trabalho, como por exemplo, Tooltip, Menu-Dropdown, Modal, Carousel, Navbar, entre outros.

Recursos como, *reset* CSS, estilo visual base para maioria das *tags*, ícones, *grids*, componentes, *plug-ins* JavaScript, e responsividade são oferecidos pelo *framework* com o objetivo de um início rápido do projeto sem perda de tempo. (MARIE, 2015).

3 QUADRO METODOLÓGICO

Neste capítulo serão abordados e descritos os procedimentos definidos e utilizados para conduzir a pesquisa. Serão apresentados o tipo de pesquisa, seu contexto, bem como os instrumentos e os procedimentos para o seu desenvolvimento.

3.1 Tipo de pesquisa

Pesquisa é um processo desenvolvido com o objetivo de se obter respostas para indagações propostas, através de conhecimentos existentes e a utilização de métodos, técnicas e procedimentos científicos. Uma pesquisa se faz necessária quando não existem respostas suficientes que satisfaçam a resolução de problemas (GIL, 2002).

Para atingir os objetivos deste trabalho, definiu-se uma pesquisa de abordagem aplicada, a qual é utilizada quando se desenvolve um produto real, com uma finalidade prática, que pode ser aplicada em determinado contexto. Conforme aponta Appolinário (2004), pesquisas aplicadas têm o objetivo de resolver problemas ou necessidades concretas e imediatas.

Aplicando esses conceitos e utilizando a pesquisa de forma aplicada por agregar maiores vantagens e demonstrar melhor os resultados obtidos, desenvolveu-se um *software* modularizado utilizando a tecnologia OSGi, com o objetivo de apresentar o paradigma da modularização e sua arquitetura, demonstrando suas características no desenvolvimento modular de *softwares*.

3.2 Contexto de pesquisa

Cada vez mais os *softwares* são utilizados em empresas, indústrias, computadores pessoais, *web* e dispositivos móveis, os quais são desenvolvidos por meio de práticas e

tecnologias existentes que auxiliam na sua criação, a não utilização de tais ferramentas torna o seu desenvolvimento e manutenção um processo desgastante e trabalhoso.

Esta pesquisa demonstra a utilização do *framework* OSGi, que possibilita o desenvolvimento do *software* em módulos, oferecendo melhor organização na sua estrutura, vantagens na manutenção e maior facilidade na expansão do mesmo, já que é construído em módulos.

Sempre que há a necessidade de adicionar, melhorar ou corrigir funcionalidades do *software*, é necessário interromper o funcionamento do mesmo para realizar o *deploy* de toda a aplicação novamente, para que então as novas funcionalidades passem a funcionar. A arquitetura de desenvolvimento modular utilizando a tecnologia OSGi propõe a resolução desses problemas, sendo possível aplicar tais funcionalidades sem comprometer todo o sistema, pois como o sistema é composto por vários módulos, passa a ser possível realizar o *deploy* de partes do sistema, parando somente o módulo necessário em vez de parar todo o sistema.

Softwares modularizados oferecem maior flexibilidade para atender empresas de diferentes áreas. Utilizando uma arquitetura modular com OSGi, os módulos desenvolvidos podem ser reutilizáveis, ou seja, os mesmos módulos podem ser utilizados em aplicações desenvolvidas para diferentes tipos de empresas. Por exemplo, um *software* é desenvolvido para determinado tipo de empresa que utiliza um módulo responsável pelo cadastro de clientes, ao desenvolver um novo *software* para outro tipo de empresa que também utilizará o cadastro de clientes, pode-se então reaproveitar o módulo de cadastro de clientes já existente. E no caso da necessidade específica de uma nova funcionalidade para outra empresa, seria necessário apenas o desenvolvimento de um novo módulo com tais funcionalidades.

Diante disso, torna-se útil o desenvolvimento de *softwares* utilizando a tecnologia OSGi, pois a mesma possibilita a criação de uma arquitetura modular que oferece vantagens na manutenção e expansão da aplicação, além de possibilitar o reaproveitamento de módulos.

3.3 Instrumentos

Durante o desenvolvimento desta pesquisa foi necessária a realização de reuniões

entre os participantes, para a obtenção e organização das informações, divisão das tarefas e desenvolvimento do *software*, pois, segundo Kioskea (2014), as reuniões são um meio para partilhar, num grupo de pessoas, um mesmo nível de conhecimento sobre um assunto ou um problema e tomar decisões coletivamente. Além disso, decisões tomadas coletivamente, com representantes das diferentes entidades interessadas, serão facilmente aceitas por todos

De acordo com Gil (2002), uma entrevista pode ser definida como uma técnica em que o pesquisador se apresenta frente ao investigado e lhe formula perguntas, com o objetivo de obtenção dos dados que interessam à pesquisa. A entrevista é, portanto, uma forma de diálogo assimétrica, em que uma das partes busca coletar dados e a outra se apresenta como fonte de informação.

Na maioria dos casos as pesquisas utilizam levantamentos bibliográficos e entrevistas com pessoas que tiveram experiências práticas com o problema pesquisado, com o propósito de esclarecer e construir hipóteses sobre o problema confrontado. (COOK, SELTZER e WRIGHTSMAN, 1987).

Para se obter mais conhecimentos acerca do tema, foi realizada uma entrevista com Filipe Portes, já que este apresenta grande conhecimento teórico e prático sobre modularização de *softwares* utilizando OSGi. Filipe Portes é graduado em Ciências da Computação pela Universidade Paulista, com mais de oito anos de experiência em Desenvolvimento e Arquitetura de Sistemas Java para *web*. Ministrou palestras sobre OSGi no TDC¹⁹ em 2012 e 2014, apresentando uma visão ampla da tecnologia, explicando conceitos do desenvolvimento modular, suas características e funcionamento, além de demonstrar exemplos práticos.

A entrevista realizou-se em 1º de agosto de 2015 com duração de 2 horas, utilizando o *software* Skype²⁰. Através deste instrumento, foram sanadas dúvidas sobre a tecnologia OSGi e conceitos da arquitetura modular. A entrevista realizada foi de grande importância, pois pôde-se compreender melhor o funcionamento da tecnologia e sua arquitetura. Ainda houve o compartilhamento de experiências, projetos e padrões de desenvolvimento, que foram utilizados para obtenção de mais conhecimentos práticos acerca do tema do trabalho. Com isso, percebeu-se falhas na arquitetura do *software* que havia sido criado, podendo então realizar as correções necessárias.

19 TDC – Abreviação para *The Developers Conference*

20 Skype – *software* gratuito que permite fazer chamadas de voz e vídeo, chat de mensagens e o compartilhamento de arquivos

3.4 Procedimentos

Nesta pesquisa desenvolveu-se uma aplicação que demonstre os conceitos de uma arquitetura modular, utilizando como principais tecnologias, a linguagem de programação Java e a especificação OSGi, que oferece suporte a modularização de *softwares*. Para o seu desenvolvimento, foi definida uma sequência de procedimentos que foram executados de forma progressiva conforme os resultados obtidos por meio dos estudos realizados em cada tecnologia.

3.4.1 Prototipação

Através dos resultados obtidos na realização de estudos sobre a especificação OSGi, desenvolveu-se protótipos a fim de verificar seu funcionamento. Foram desenvolvidos diferentes tipos de protótipos, com o objetivo de definir como seria desenvolvida uma aplicação que demonstrasse a modularização de *softwares* e seu funcionamento.

Desenvolveram-se protótipos que podem ser executados no ambiente do tipo *desktop* e *web*. De acordo com as informações e resultados obtidos, definiu-se que a aplicação desenvolvida para exemplificar a modularização de *software* seria do tipo *web*.

A especificação OSGi é implementada por vários *frameworks*. Para os protótipos desta pesquisa utilizou-se o Equinox e o Apache Felix. Desta forma, definiu-se como implementação a ser utilizada para o desenvolvimento o *framework* Apache Felix, devido a sua excelente integração com o servidor de aplicação GlassFish. Sendo assim, definiu-se também, a utilização do mesmo, completando assim o escopo *back-end* para o desenvolvimento de uma aplicação *web*.

Com o desenvolvimento dos protótipos e embasado nos estudos realizados, observou-se que a tecnologia OSGi oferece suporte a projetos do tipo Java *Application*, *Web Application* e *Enterprise* JavaBeans. Com isso, constatou-se a possibilidade de desenvolver módulos de diferentes tipos, que interagem entre si através de interfaces bem definidas, na qual são disponibilizadas como serviços ou exportadas dentro do contexto OSGi.

Foram desenvolvidos também protótipos de módulos responsáveis por disponibilizar a interface gráfica com o usuário, utilizando as tecnologias HTML, JavaScript, Angular JS, CSS e Bootstrap.

A prototipação foi um procedimento muito importante para o desenvolvimento deste trabalho, pois além de se comprovar o funcionamento prático das teorias estudadas, observou-se que o desenvolvimento modular requer um planejamento complexo de sua arquitetura, proporcionando desacoplamento dos módulos e alta coesão, ou seja, os módulos podem ser desinstalados, parados e atualizados em tempo de execução sem afetar outros módulos, além de definir responsabilidades específicas para cada módulo.

3.4.2 Definição do *software*

Após estudos e criação de protótipos, definiu-se uma aplicação básica contendo cinco módulos, que demonstra uma arquitetura modular utilizando a tecnologia OSGi através do *framework* Apache Felix. O objetivo do *software* é apenas demonstrar o funcionamento da arquitetura assim como características da tecnologia OSGi.

- **Módulo Util:** responsável por conter operações globais, armazenar e disponibilizar os recursos *web* (CSS, JavaScript, bibliotecas, imagens, etc.) compartilhados para todos os outros módulos;
- **Módulo Home:** contém a tela inicial do sistema após a autenticação do usuário.
- **Módulo Usuário:** responsável por controlar os cadastros de usuários e autenticação dos mesmos no sistema;
- **Módulo Clientes:** responsável por controlar os cadastros de clientes, com integração ao módulo financeiro;
- **Módulo Financeiro:** responsável por controlar os lançamentos de contas a pagar e a receber, com integração ao módulo de clientes;
- **Módulo Log:** responsável por registrar logs do sistema em arquivos, com integração entre todos os outros módulos.
- **Módulo Data Source:** responsável por fornecer e controlar acesso ao banco de dados,

com integração entre todos os outros módulos.

A aplicação foi definida para execução no ambiente *web*, sendo implantada no servidor de aplicação GlassFish. Isso se justifica devido à grande flexibilidade que os sistemas *web* oferecem, podendo ser executado em qualquer sistema operacional, assim como qualquer dispositivo que tenha suporte ao *browser*.

3.4.3 Modelagem do banco de dados

Para a aplicação desenvolvida faz-se necessário o uso de um banco de dados para armazenar os dados utilizados pelo sistema. Dessa forma foi necessário criar uma modelagem do banco de dados conforme demonstra a Figura 8.

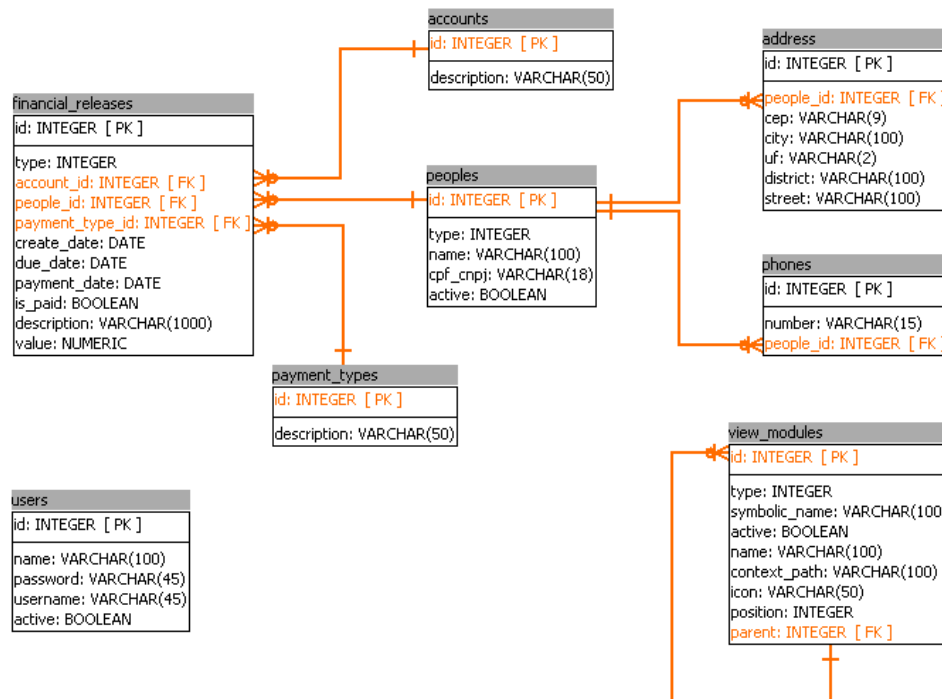


Figura 8 – Modelagem do banco de dados. **Fonte:** Elaborado pelos autores.

Os dados salvos no banco de dados não influenciam no funcionamento da tecnologia OSGi. São somente utilizados para o controle e registro de informações utilizadas na aplicação pelo usuário.

3.4.4 Modelagem da arquitetura dos módulos

A arquitetura de um *software* é uma das principais partes do seu desenvolvimento. Dessa maneira, desenvolveu-se uma arquitetura modular que fornece aos módulos flexibilidade para que possam ser desinstalados, parados e atualizados enquanto o restante do sistema está em funcionamento.

Modelou-se uma arquitetura em que os módulos do sistema são formados por outros módulos menores. Normalmente, cada módulo do *software* está composto por outros três módulos, são eles: módulo de visão, responsável por conter as telas de interação com o usuário. O módulo API, que contém funcionalidades e interfaces que são compartilhadas e expostas como serviços, e o módulo *core*, que contém a implementação das interfaces do módulo API e a lógica de negócio da aplicação.

Com os módulos dispostos dessa maneira, o único módulo que gera dependência para outros módulos é o API, com isso o módulo de visão e *core* podem ser desinstalados, parados e atualizados sem comprometer o restante do *software*. Definiu-se então, que o módulo API é a base principal para os módulos de visão e *core*, não podendo ser desinstalado do sistema, a não ser que outro módulo venha a substituí-lo com as mesmas interfaces.

Essa arquitetura pode ser entendida melhor de acordo com a Figura 9 que demonstra a arquitetura de um módulo do sistema, além de suas camadas.

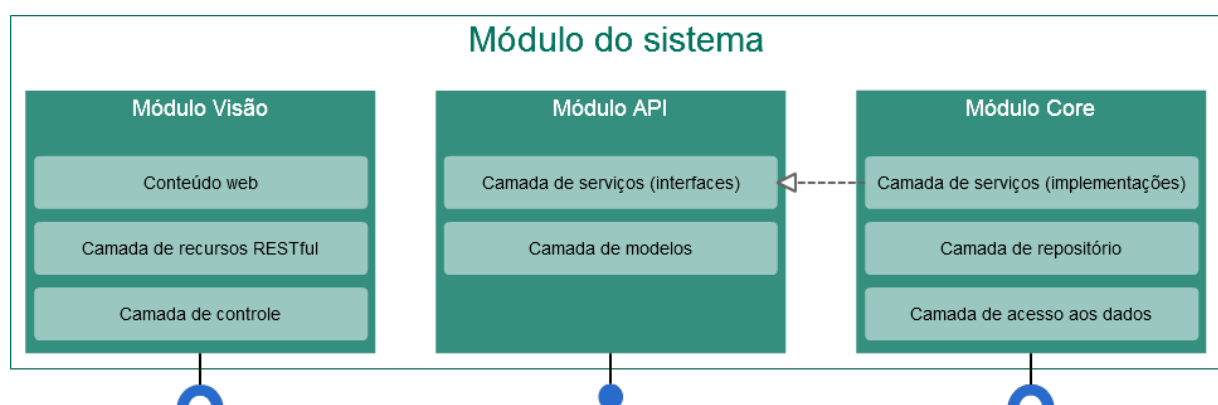


Figura 9 – Arquitetura de um módulo do sistema. **Fonte:** Elaborado pelos autores.

As camadas podem se diferenciar entre os módulos do sistema ou dependendo da necessidade da aplicação. Porém para o *software* desenvolvido, essa foi a estrutura comum de

camadas para os módulos. A Figura 10 representa os serviços disponibilizados pelos módulos API.



Figura 10 – Representação dos serviços disponibilizados pelos módulos. **Fonte:** Elaborado pelos autores.

Os módulos que consomem os serviços disponibilizados são representados conforme a Figura 11.



Figura 11 – Representação dos módulos que consomem serviços. **Fonte:** Elaborado pelos autores.

A arquitetura dos módulos foi desenvolvida com base nos estudos, práticas pesquisadas e de acordo com necessidade da aplicação, além das experiências obtidas na entrevista realizada neste trabalho. Porém isso pode mudar dependendo da aplicação que se deseja desenvolver. A Figura 12 demonstra como foi definida a arquitetura de todos os módulos que compõem o *software* desenvolvido.

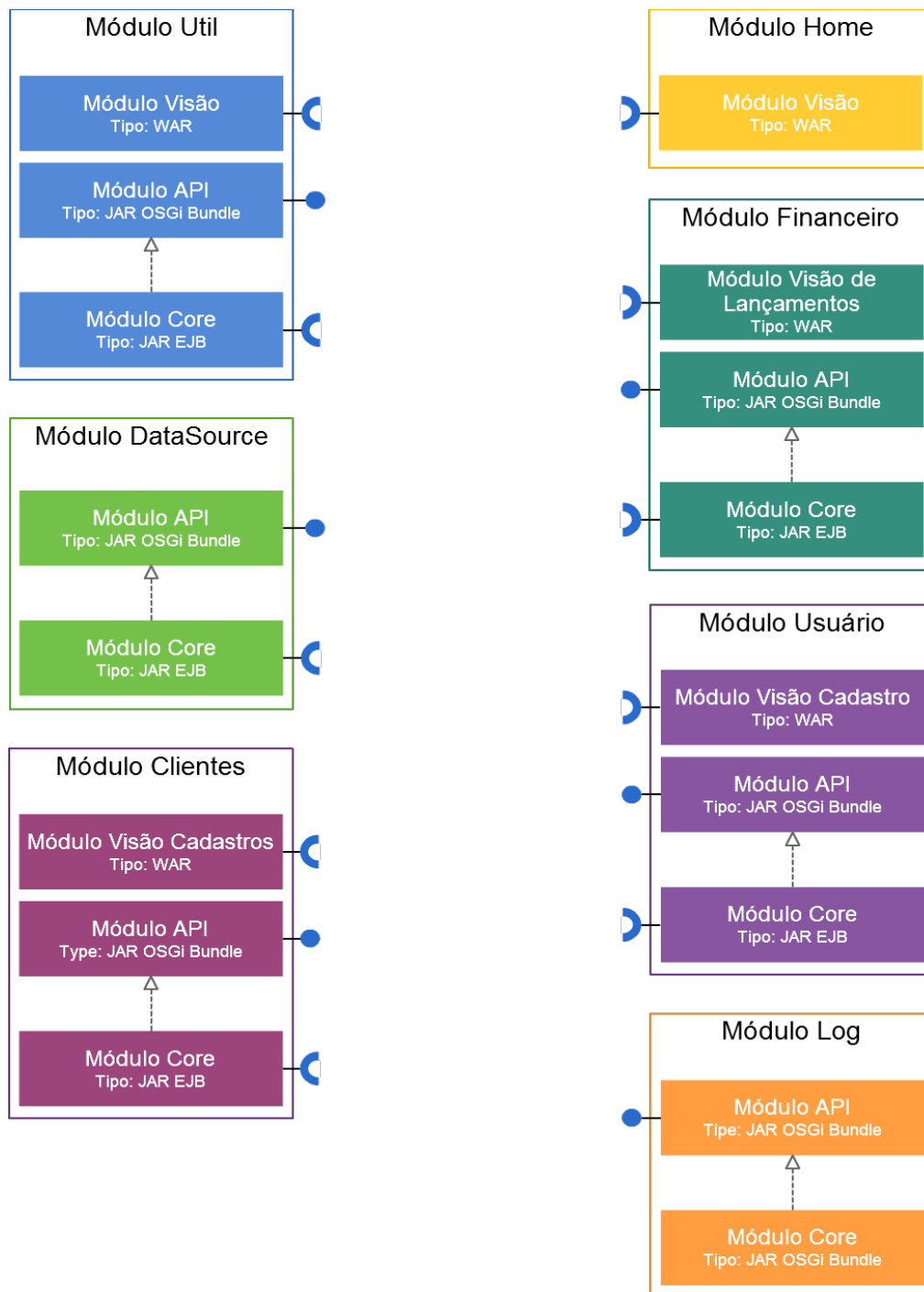


Figura 12 – Arquitetura dos módulos do sistema. **Fonte:** Elaborado pelos autores.

Através da modelagem dessa arquitetura, foi possível desenvolver um *software* em que os módulos não dependam diretamente da implementação. Isso se fez devido à criação de interfaces bem definidas que estão nos módulos APIs. Resultando em um *software* desacoplado e também de alta coesão devido a cada módulo ter sua responsabilidade bem definida.

3.4.5 Desenvolvimento

O *software* desenvolvido está separado por duas partes que possuem responsabilidades específicas, o *back-end* e *front-end*. Os módulos foram divididos de forma que estas partes fiquem bem definidas. O *back-end* é composto pelos módulos que contém regras de negócio, recursos e serviços. Enquanto o *front-end* é responsável por realizar uma interface entre o sistema e o usuário, utilizando as funcionalidades do *back-end*.

O *front-end* da aplicação é composto pelas telas desenvolvidas utilizando as tecnologias HTML, CSS, Bootstrap, JavaScript, Angular JS e JQuery. Essa parte do sistema está disposta nos módulos de visão que são projetos do tipo *Web Application*.

Como dito anteriormente, basicamente cada módulo do sistema é composto por outros três módulos menores, o módulo de visão, API e *core*. No módulo de visão, estão os recursos RESTful e controle de fluxo. No módulo API estão definidos as interfaces que são expostas como serviços e funcionalidades, ambas estão disponíveis para qualquer outro módulo. E por fim, no módulo *core* está a implementação das interfaces, assim como toda lógica de negócio e controle do módulo. Todos esses fatores compõem a parte *back-end* da aplicação.

Os módulos foram desenvolvidos utilizando a IDE de desenvolvimento NetBeans 8.0.2 acompanhado do padrão Maven que propõe uma estrutura para cada tipo de projeto, além de possibilitar que projetos sejam criados em uma estrutura hierárquica que possibilita criar um projeto principal que controle outros projetos que o compõe. O Maven já vem integrado ao NetBeans, desta forma não foi necessária nenhuma configuração para a utilização do mesmo.

A estrutura oferecida pelo Maven é muito útil para uma aplicação modular, pois, como cada módulo é um projeto, o mesmo permite melhor controle da hierarquia dos módulos, além de todas as configurações de geração dos projetos estarem definidas em um projeto principal.

A estrutura do projeto está separada em um projeto principal do tipo POM *Project*, que contém todos os outros projetos. O mesmo é composto por outros projetos do mesmo tipo, que representam os módulos do sistema, que por fim, são compostos pelos projetos do tipo *Web Application* (módulo de visão), *Java Application* (módulo de API) e *Enterprise JavaBeans* (módulo *core*).

O projeto do tipo POM *Project* foi criado selecionando a opção **Maven** → **POM**

Project após seleccionar a opção **File** → **New Project** no menu principal do NetBeans, como demonstrado nas Figuras 13 e 14.

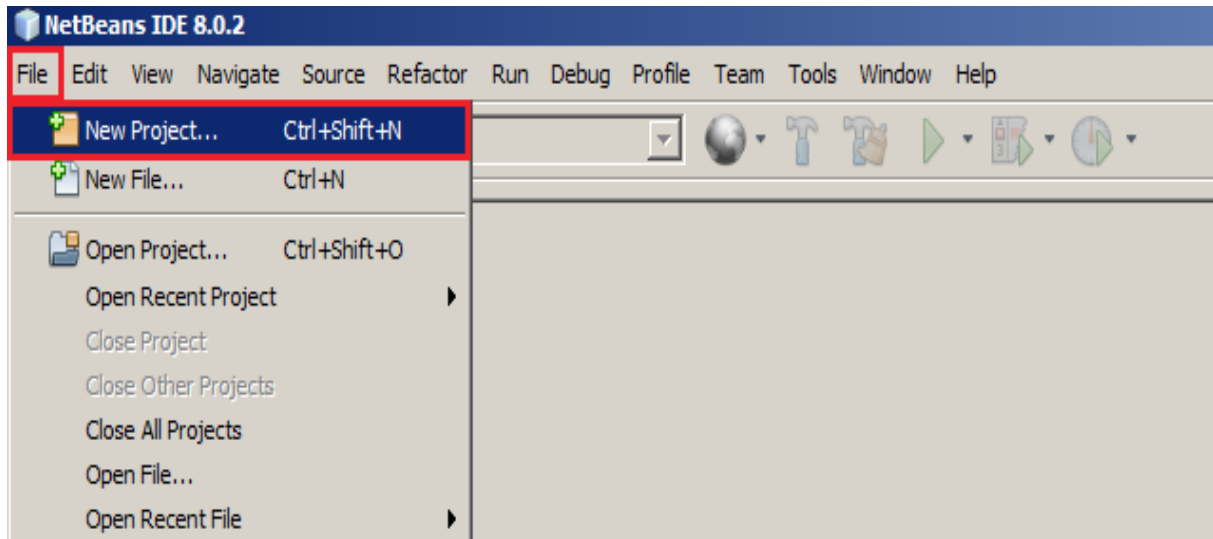


Figura 13 – Opção para criar projeto. **Fonte:** Elaborado pelos autores.

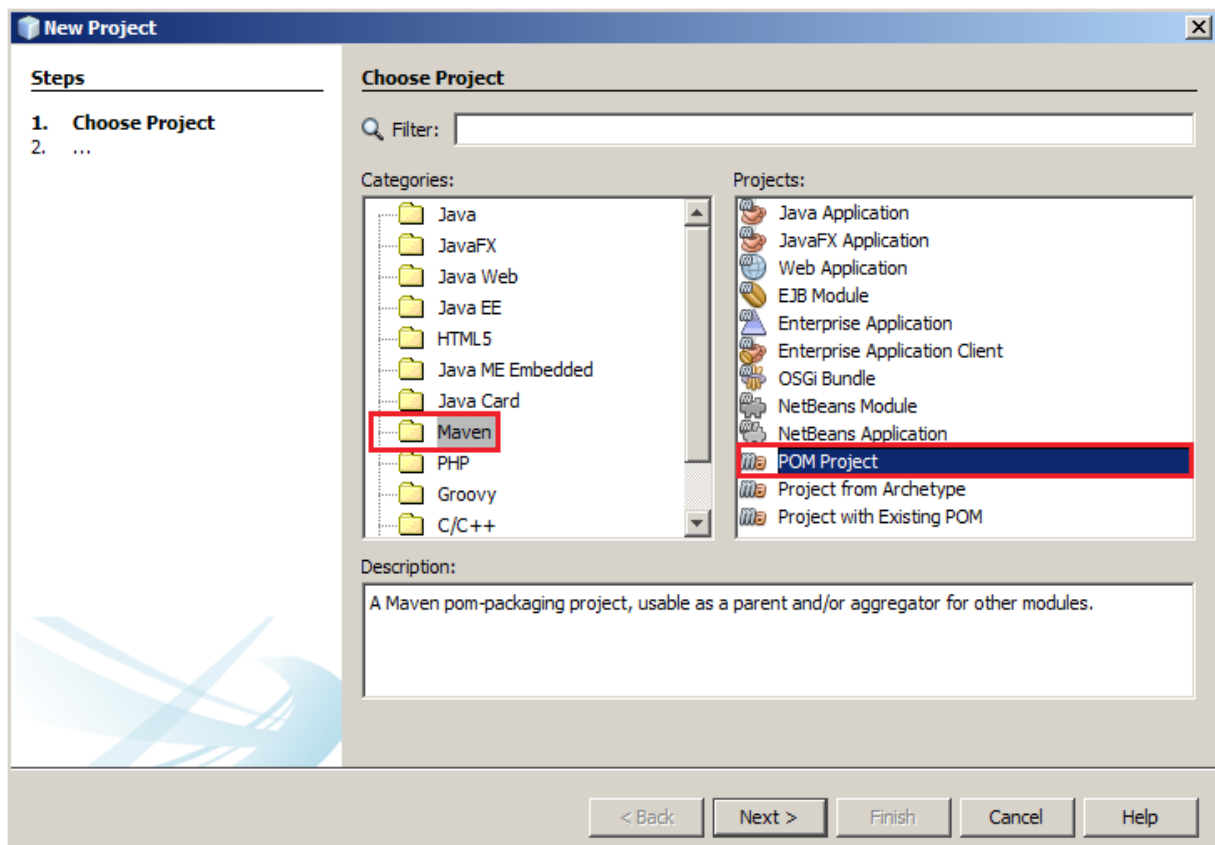


Figura 14 – Tela para escolha do tipo do projeto como POM Project. **Fonte:** Elaborado pelos autores.

Em seguida, como demonstra a Figura 15, foi definido o nome do projeto e escolhido

o local onde o mesmo seria salvo. Além dessas informações, foi definido também o **Artifact Id, Group Id, Version e Package**.

- **Artifact Id:** nome único para o projeto dentro do contexto do Maven;
- **Group Id:** grupo definido para os projetos;
- **Version:** versão definida para o projeto, que será a versão do módulo;
- **Package:** nome inicial para a estrutura de pacotes do projeto. Esta informação é opcional para projetos do tipo *POM Project*.

As opções descritas acima são utilizadas pelo Maven para controlar o projeto, além de serem utilizadas para geração final do projeto de cada módulo.

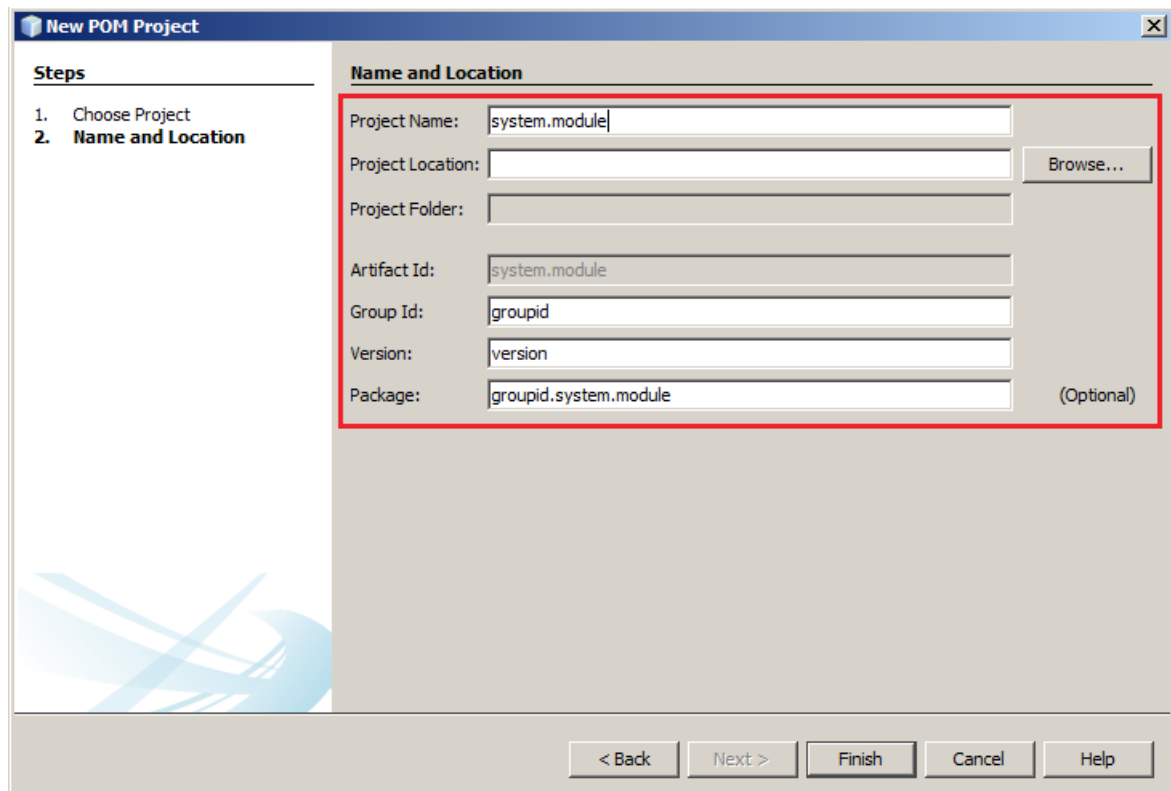


Figura 15 – Tela de informações do novo projeto. Fonte: Elaborado pelos autores.

Os projetos que representam os módulos do sistema, também são do tipo *POM Project* e foram criados através da opção **Modules → Create New Module** localizada no projeto criado anteriormente, conforme é demonstrado na Figura 16. Após escolhida essa opção, basta seguir os procedimentos da Figura 14 e 15.

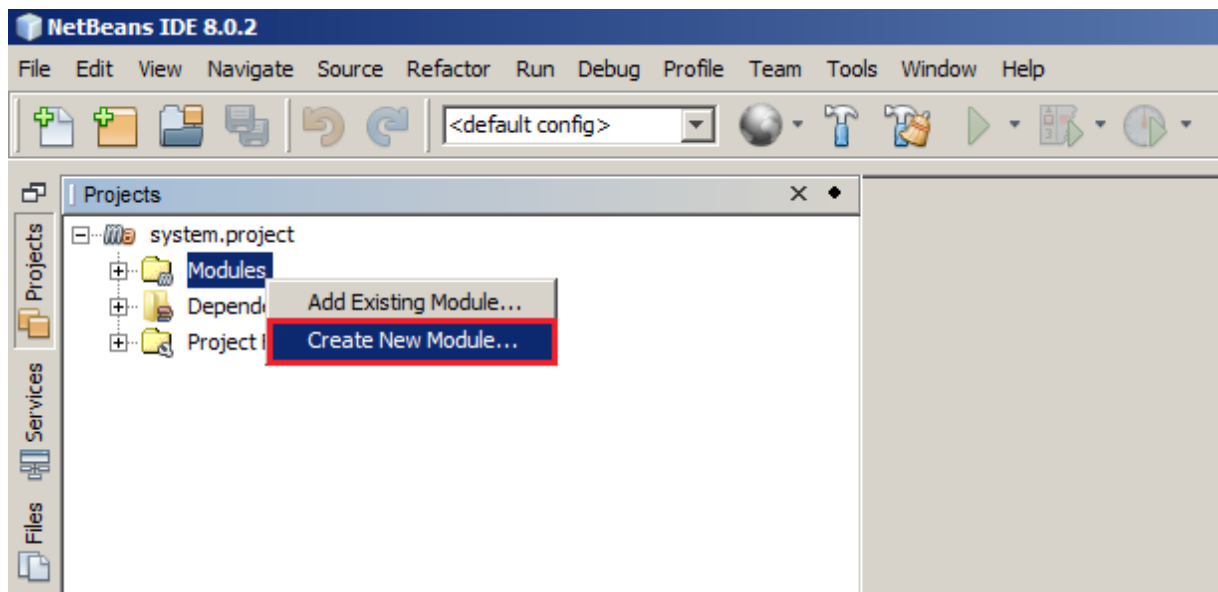


Figura 16 – Criação de novo projeto como módulo. **Fonte:** Elaborado pelos autores.

Após fazer as etapas descritas até aqui, o projeto ficou com uma estrutura conforme demonstrada na Figura 17.

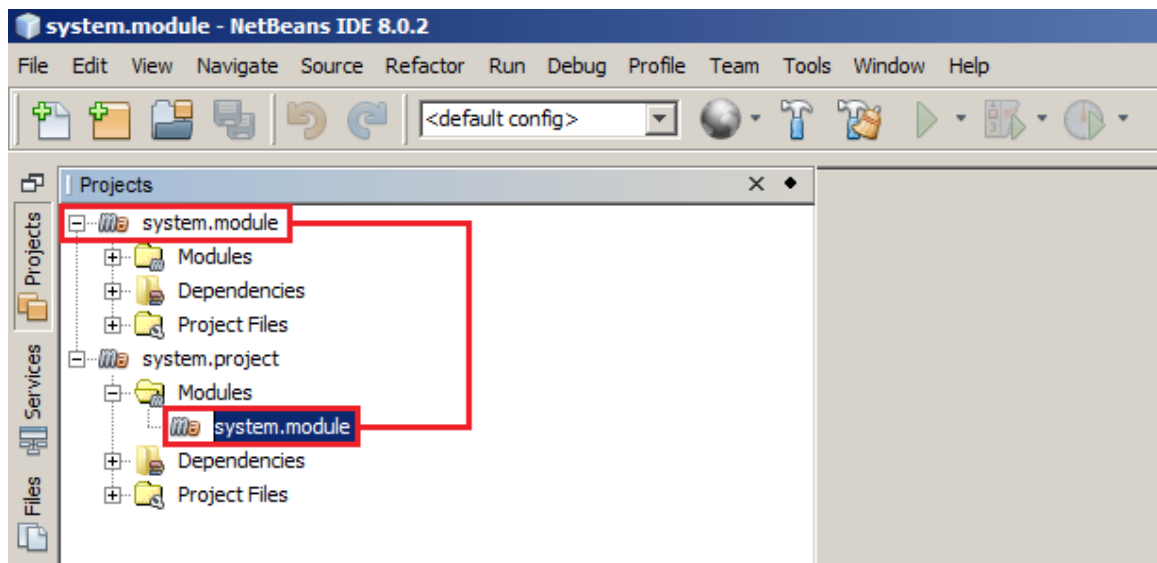


Figura 17 – Estrutura do projeto principal composto por outro projeto. **Fonte:** Elaborado pelos autores.

A criação dos projetos do tipo *Web Application* foi realizada através da opção **Modules** → **Create New Module** no projeto que representa os módulos do sistema, conforme demonstrado na Figura 18.

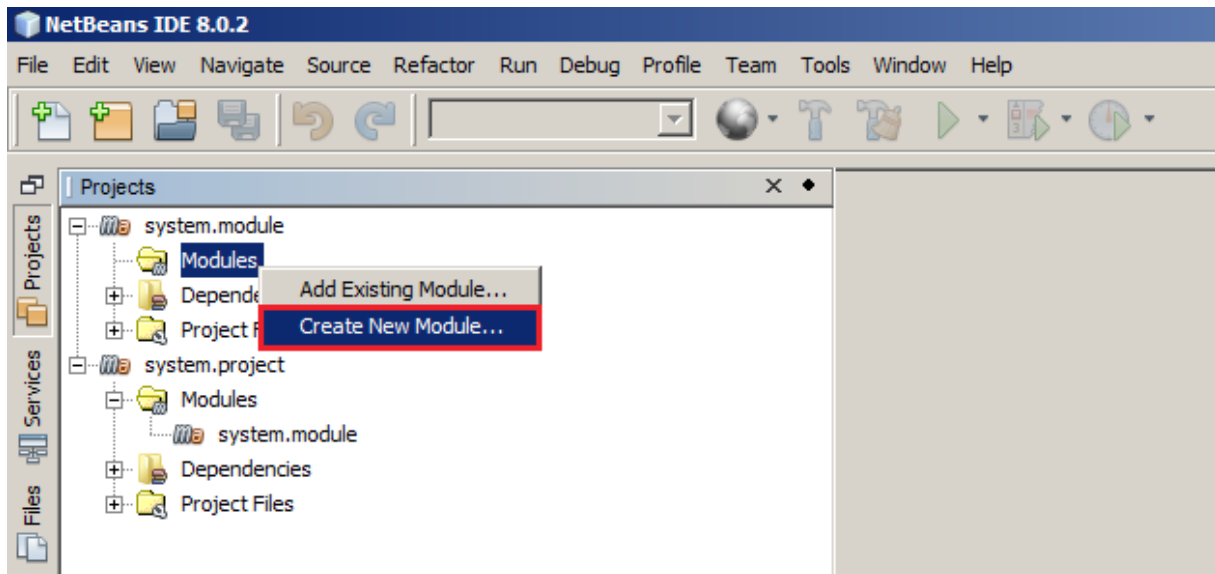


Figura 18 – Criação de novo projeto como módulo. Fonte: Elaborado pelos autores.

Em seguida deve ser escolhida a opção **Maven** → **Web Application** conforme a Figura 19.

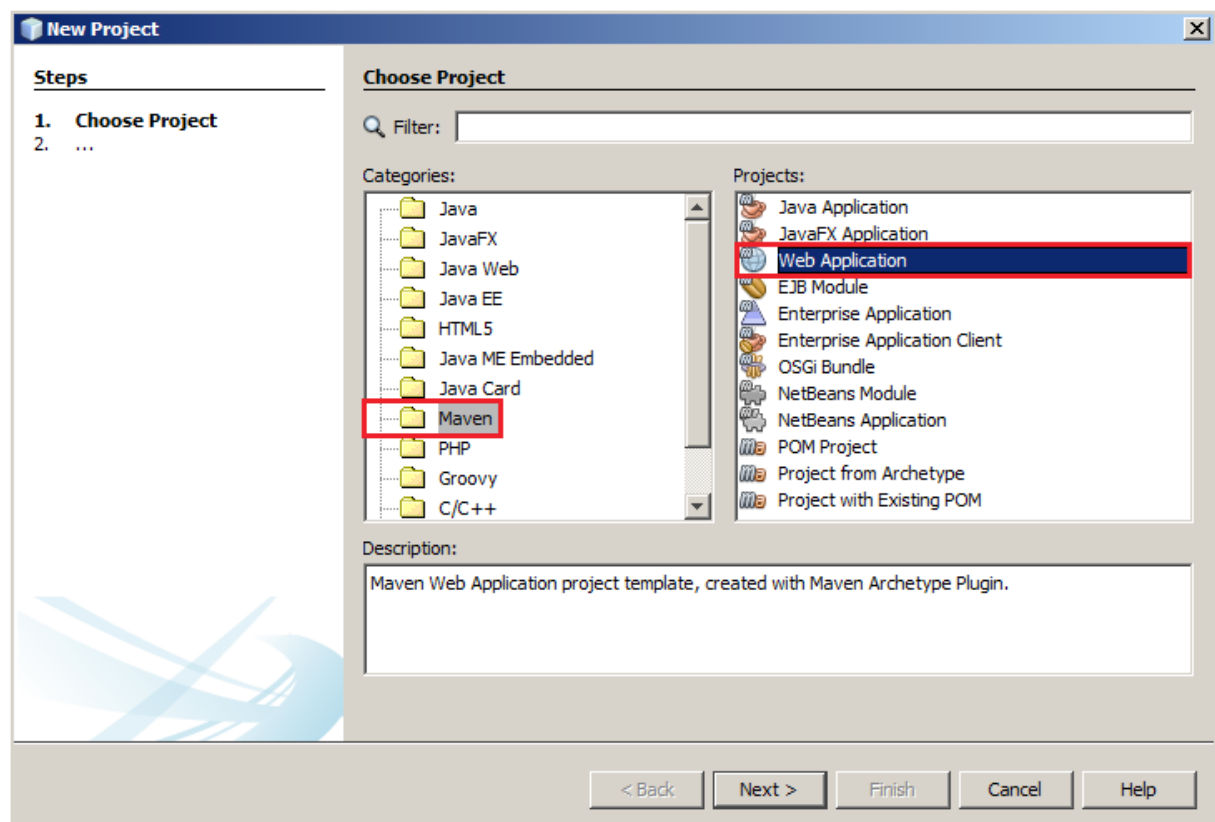


Figura 19 – Tela para escolha do tipo do projeto como Web Application. Fonte: Elaborado pelos autores.

Após isto, basta definir as informações do novo projeto conforme a Figura 15 e, por

fim, é necessário definir o GlassFish como servidor de aplicação na opção **Server** e escolher Java EE 7 Web para a opção **Java EE Version** como mostra a Figura 20.

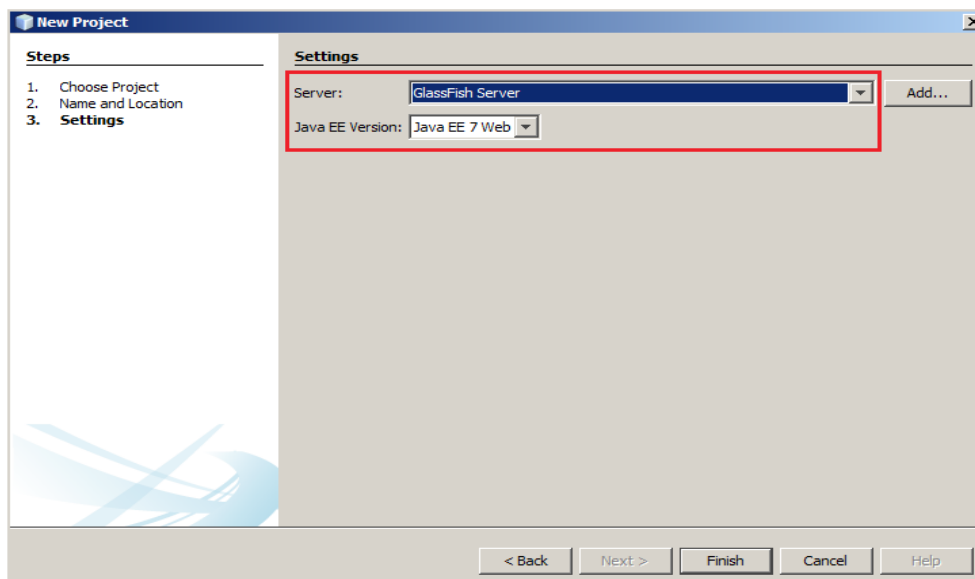


Figura 20 – Tela de configuração do Server e Java EE Version. **Fonte:** Elaborado pelos autores.

Para a criação dos projetos do tipo *Java Application* basta seguir os passos da Figura 18, em seguida escolher a opção **Maven** → **Java Application** ou **Maven** → **OSGi Bundle** conforme demonstra a Figura 21. Após isto, basta definir as informações do novo projeto conforme demonstrado na Figura 15.

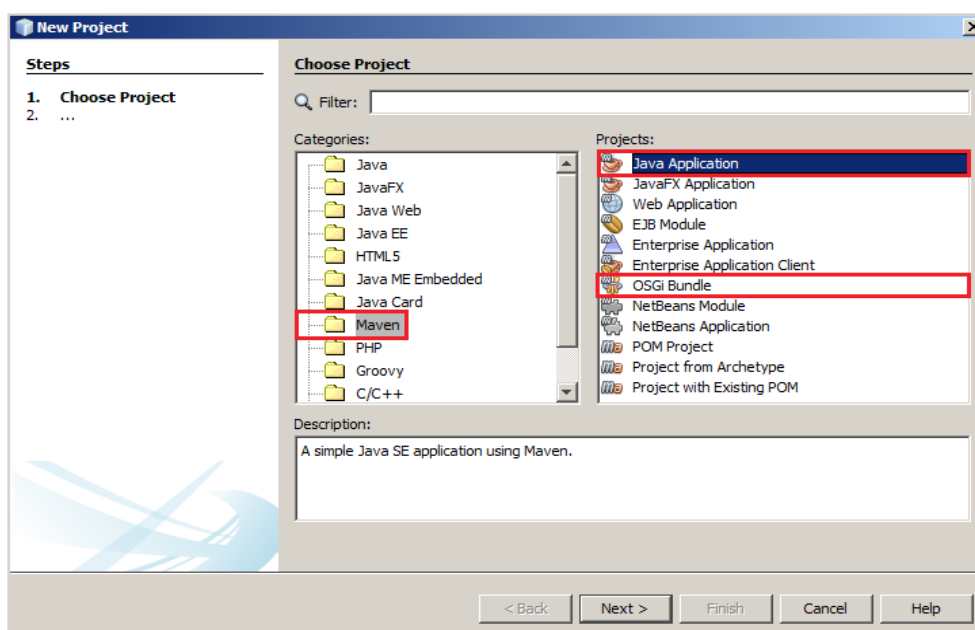


Figura 21 – Tela para escolha do tipo do projeto como *Java Application* ou *OSGi Bundle*. **Fonte:** Elaborado pelos autores.

Os projetos do tipo *Enterprise JavaBeans* foram criados conforme os passos demonstrados na Figura 18, em seguida escolhida a opção **Maven** → **EJB Module** conforme Figura 22. Em seguida, basta seguir os passos da Figura 15 e Figura 20.

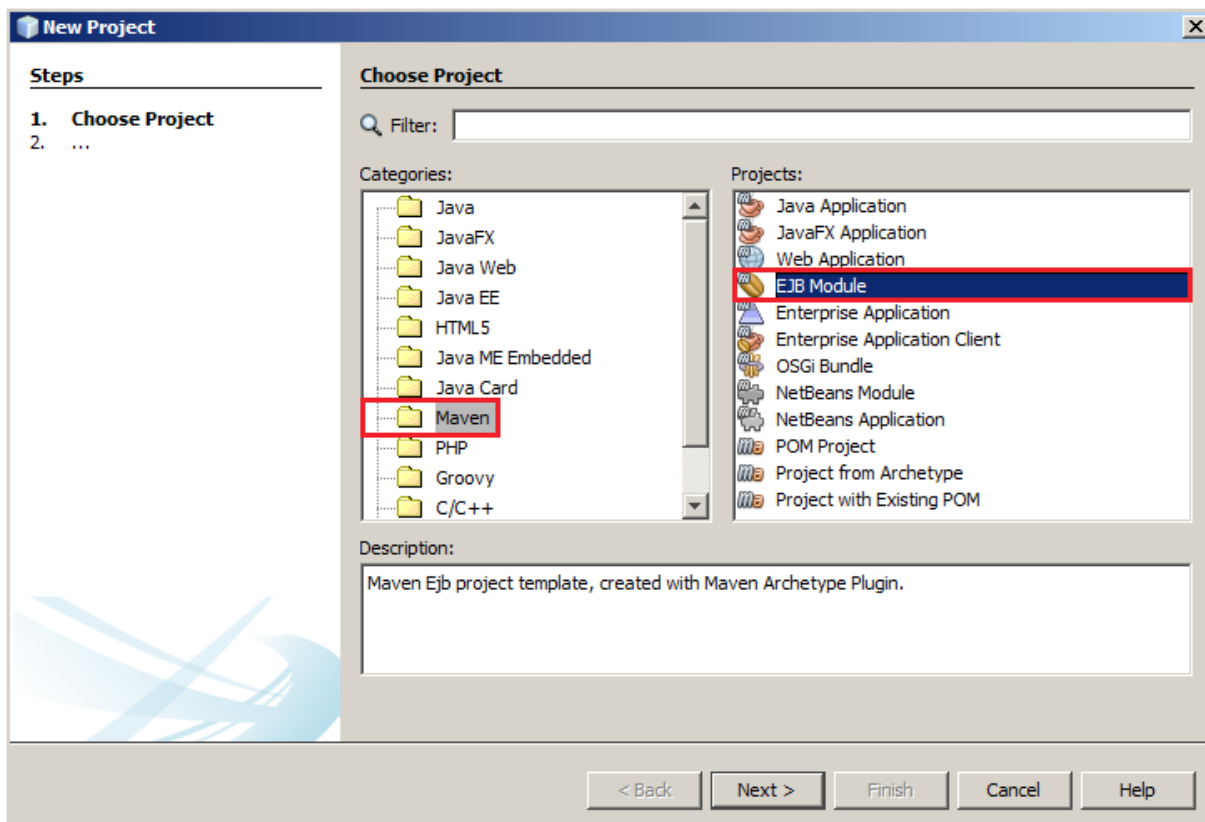


Figura 22 – Tela para escolha do tipo do projeto como Enterprise JavaBeans. **Fonte:** Elaborado pelos autores.

Após realizadas todas essas etapas, a estrutura do projeto ficou composta por um projeto do tipo *POM Project* que representa todo o sistema. O mesmo está composto por outros projetos do tipo *POM Project* que representam os módulos do sistema, que estão compostos pelos projetos do tipo *Web Application*, *Java Application* e *Enterprise JavaBeans* que são respectivamente os módulos de visão, API e *core*. Essa estrutura é demonstrada na Figura 23.

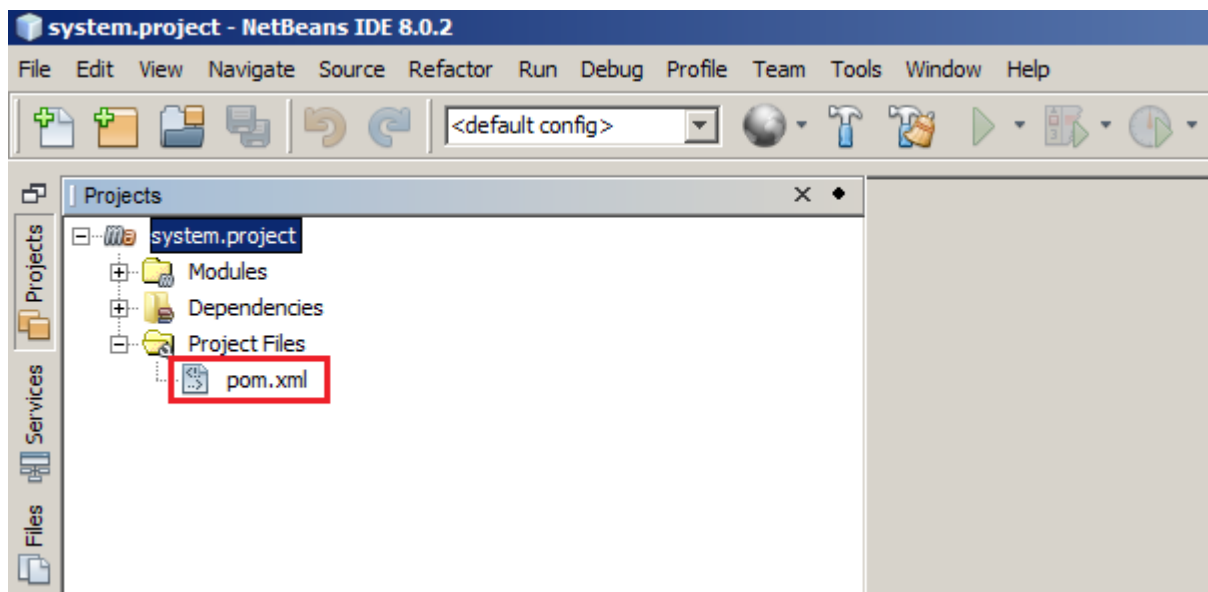


Figura 24 – Localização do arquivo pom.xml na estrutura do projeto. Fonte: Elaborado pelos autores.

Cada projeto criado possui seu arquivo **pom.xml** que contém suas propriedades, porém como o Maven disponibiliza uma hierarquia que possibilita que propriedades do arquivo **pom.xml** do projeto principal possam ser utilizadas pelos projetos que o compõe, as configurações principais estão configuradas no mesmo e disponibilizadas para os outros projetos.

Para o projeto do tipo *POM Project* que representa todo o sistema, o arquivo **pom.xml** foi dividido em três partes: as informações principais do projeto, que estão demonstradas na Figura 25, as configurações de dependências demonstradas, na Figura 26 e as configurações de *plugins* e construção do projeto, conforme demonstrado nas Figuras 27, 28, 29 e 30.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
  <modelVersion>4.0.0</modelVersion>

  <groupId>groupid</groupId>
  <artifactId>system.project</artifactId>
  <version>version</version>
  <packaging>pom</packaging>

  <modules>
    <module>system.module</module>
  </modules>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
```

Figura 25 – Informações principais do arquivo pom.xml. Fonte: Elaborado pelos autores.

As informações destacadas pelo retângulo vermelho de número 1 na Figura 25 são referentes aos dados informados na criação do projeto conforme demonstrado na Figura 15. O retângulo número 2 destaca a *tag* **<modules>** que identifica os projetos que compõe esse projeto.

Na Figura 26, conforme destacado pelo retângulo número 3, inicia-se o gerenciamento das dependências do Maven através da *tag* **<dependencyManagement>**. Dependências são bibliotecas e *frameworks* normalmente disponibilizadas por terceiros que são utilizados dentro do projeto, além dos módulos do sistema que foram desenvolvidos, que após construídos passam a se tornar dependências.

O Maven possui um repositório onde estão armazenados todas as bibliotecas e *frameworks*, com isso ao informarmos no arquivo **pom.xml** alguma dependência conforme mostra o retângulo número 4 da Figura 26, o Maven se encarrega de fazer o *download* e registrar tais dependências no projeto. Quando se trata de módulos que foram criados, o Maven busca essa dependência dentro da arquitetura do projeto e registra o mesmo.

Ao utilizarmos a *tag* **<dependencyManagement>**, conforme destacado no retângulo número 3 da Figura 26, é indicado ao Maven que essas dependências serão utilizadas por outros projetos, desta maneira, o mesmo não fará o *download* dessas dependências para esse projeto, mas sim para o projeto que indicarem essa dependência em seu **pom.xml** e estiverem dentro da hierarquia desse projeto. Essa *tag* é muito útil devido ao versionamento das dependências, pois é definida somente uma vez a versão de cada dependência através da *tag* **<version>**, com isso, o projeto que for utilizar essa dependência não precisará informar sua versão. Desta maneira, caso seja necessário trocar a versão de uma dependência, basta trocar a versão da mesma no **pom.xml** do projeto principal que sua versão será alterada em todos os outros projetos que compõe este projeto.

A *tag* **<scope>** define como a dependência será fornecida ao projeto no ambiente de execução, o valor **provided** indica que o servidor de aplicação onde o projeto será implantado é quem fornecerá essa dependência. Com o desenvolvimento modular praticamente todas as dependências têm a *tag* **<scope>** com o valor **provided**, pois são dependências que já devem estar instaladas no *framework* Apache Felix que está integrado com o servidor de aplicação GlassFish.

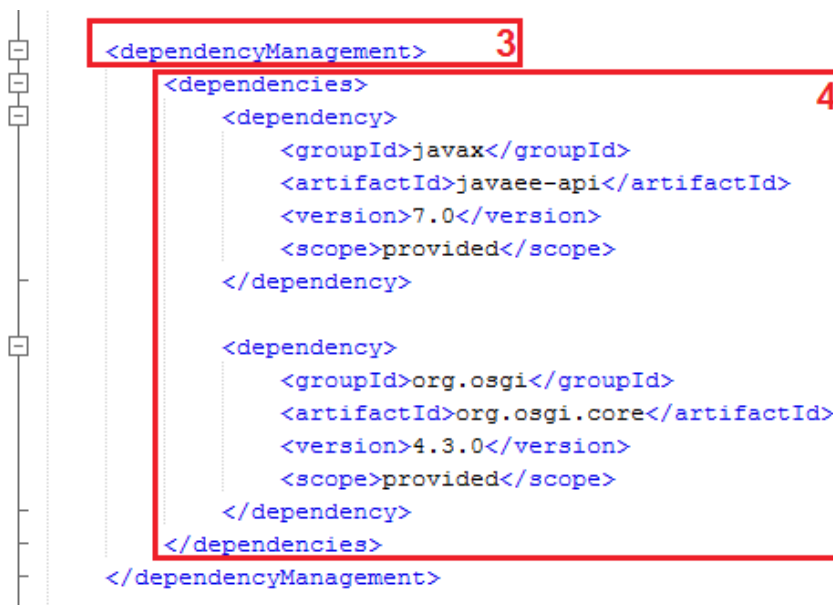


Figura 26 – Informações do gerenciamento de dependências. Fonte: Elaborado pelos autores.

O Maven também possibilita a utilização de *plugins* que são utilizados para executarem tarefas durante a construção do projeto. Estão sendo utilizados 5 *plugins* para a geração dos projetos.

Os *plugins* podem ser gerenciados de forma parecida com o gerenciamento de dependências explicado anteriormente. Ao informar a tag `<pluginManagement>`, conforme destacada com o retângulo número 5 da Figura 27, o Maven passa a gerenciar todas os plugins dentro da mesma, porém os *plugins* ainda não estão disponíveis para os outros projetos. Para que isso aconteça é necessário informar uma tag `<plugins>` fora da tag `<pluginManagement>` conforme destacado com o retângulo de número 11 na Figura 30. O *plugin* informado na tag `<plugins>`, é uma referência ao *plugin* **maven-bundle-plugin** que está dentro da tag `<pluginManagement>`. Ao fazer isso somente este *plugin* fica disponível para os demais projetos.



Figura 27 – Configurações de *plugins* (parte 1). Fonte: Elaborado pelos autores.

O *plugin* **maven-bundle-plugin** destacado no retângulo número 6 da Figura 27, é o principal *plugin* utilizado no desenvolvimento da aplicação, pois é responsável por construir o projeto como um módulo, ou seja, construir um *bundle* que é reconhecido como um módulo no contexto OSGi.

Neste *plugin* são configurados através da tag **<supportedProjectTypes>**, quais os tipos de projetos que podem ser gerados no momento da construção do pacote. Como explicado anteriormente, a aplicação está composta por projetos do tipo *Web Application*, *Java Application* e *Enterprise JavaBeans*, para que esses projetos sejam gerados corretamente, é necessário então que essa tag tenha os valores **bundle**, **ejb**, **jar** e **war**.

Além de configurar os tipos de projetos que serão gerados por esse *plugin*, é necessário que estejam configurados no arquivo **pom.xml** os *plugins* referentes a cada tipo de

projeto. Para projetos do tipo *Enterprise JavaBeans* é utilizado o *plugin maven-ejb-plugin*, conforme demonstrado no retângulo número 7 da Figura 28. Para os projetos do tipo *Java Application* ou *OSGi Bundle* é utilizado o *plugin maven-jar-plugin*, destacado pelo retângulo número 8. E para os projetos do tipo *Web Application* é utilizado o *plugin maven-war-plugin*, conforme destaca o retângulo 9 da Figura 29.



Figura 28 – Configurações de *plugins* (parte 2). Fonte: Elaborado pelos autores.



Figura 29 – Configurações de *plugins* (parte 3). Fonte: Elaborado pelos autores.

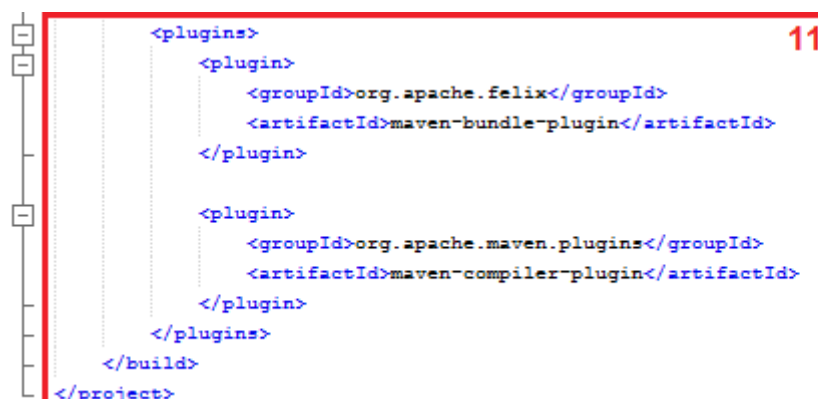


Figura 30 – Configurações de *plugins* (parte 4). Fonte: Elaborado pelos autores.

Durante a construção do projeto, o *plugin* **maven-bundle-plugin** gera o arquivo MANIFEST.MF que contém as informações necessárias para que o *framework* Apache Felix possa disponibilizar o projeto como um módulo. Devido a esse arquivo possuir propriedades específicas de cada projeto, foi necessário realizar outra configuração nesse *plugin*, conforme mostra o retângulo número 6 da Figura 27, uma instrução de inclusão do arquivo **osgi.properties** é configurada através da *tag* `<_include>`. Desta maneira, durante a

construção do projeto, o *plugin* busca esse arquivo na raiz do projeto e coloca essas informações no arquivo MANIFEST.MF. A Figura 31 demonstra o conteúdo do arquivo.

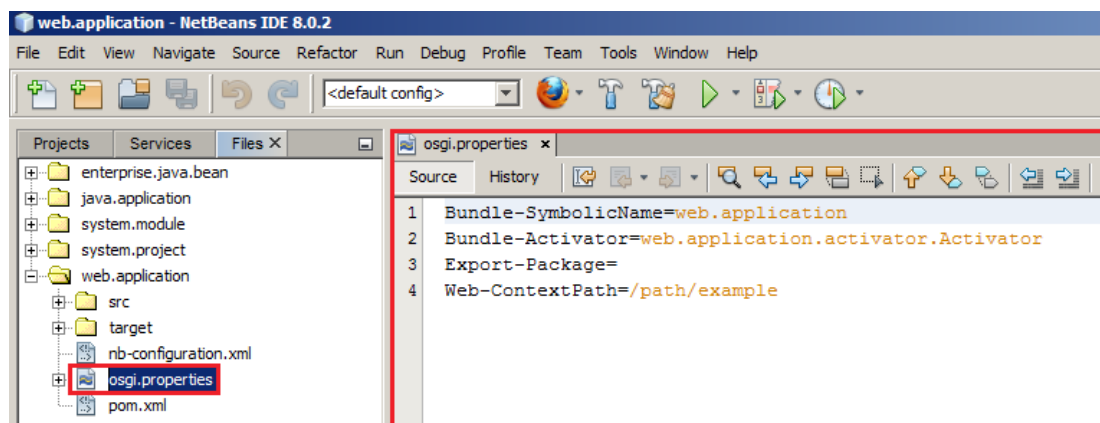


Figura 31 – Arquivo osgi.properties. Fonte: Elaborado pelos autores.

O arquivo **osgi.properties** deve ser adicionado manualmente na raiz de cada projeto informando cada propriedade específica de acordo com seu tipo.

- **Bundle-SymbolicName:** é indicado o nome definido para o módulo;
- **Bundle-Activator:** é indicado o caminho da classe que implementa a interface BundleActivator.java do *framework* OSGi;
- **Export-Package:** é indicado quais pacotes são expostos por esse módulo;
- **Web-ContextPath:** é indicado o caminho da URL de acesso caso o módulo seja do tipo *Web Application*.

A utilização do **maven-bundle-plugin** auxiliou muito na geração dos módulos, pois além das configurações demonstradas anteriormente, o *plugin* se encarrega de preencher diversas outras propriedades do arquivo MANIFEST.MF, o que teria sido muito trabalhoso se realizado manualmente.

Após configurar adequadamente toda a estrutura do projeto, definiu-se as interfaces que são expostas como serviços entre os módulos e suas implementações. A seguir é demonstrado as interfaces e implementações do Módulo Log do sistema para um melhor entendimento. No módulo API foram definidas as interfaces que funcionam como serviços para os outros módulos. A Figura 32 demonstra a interface que representa o serviço do Módulo Log e seus métodos.

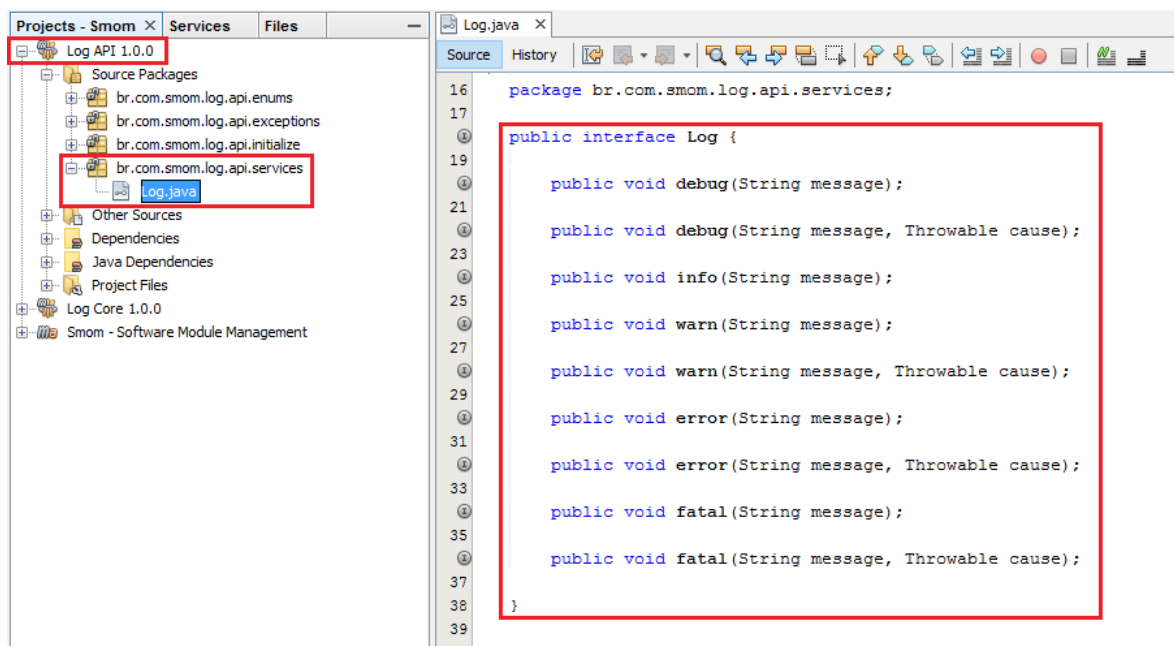


Figura 32 – Serviço do Módulo Log API. Fonte: Elaborado pelos autores.

A Figura 32 demonstra a interface **Log.java** composta por seus métodos. Essa interface faz parte do **Módulo Log API** e é implementada pela classe **LogService.java** que está no **Módulo Log Core** conforme demonstra a Figura 34.

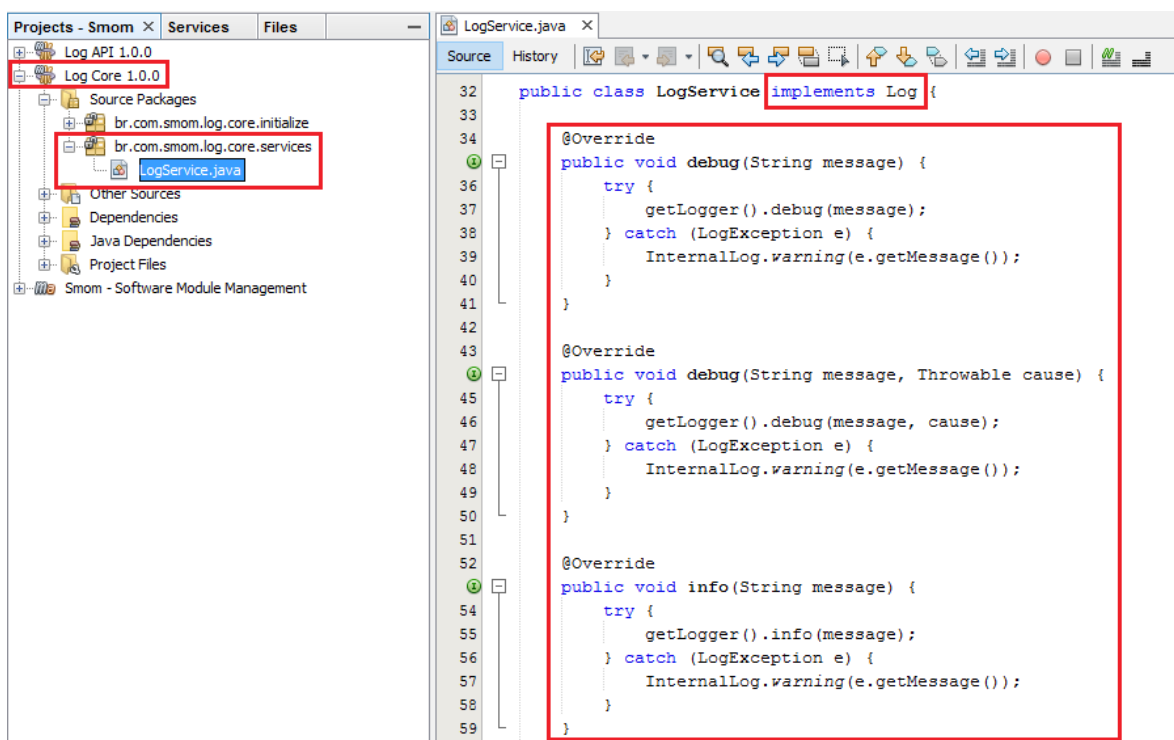


Figura 33 – Implementação do serviço do Módulo Log API. Fonte: Elaborado pelos autores.

Nesse módulo está implementada toda a lógica para a geração de *logs* no sistema. Porém os outros módulos do sistema conhecem somente o **Módulo Log API**, dessa forma, a implementação fica independente dentro do sistema, possibilitando assim que a mesma possa ser parada, desinstalada e atualizada sem comprometer o restante da aplicação. O restante dos módulos do sistema seguem a mesma ideia e estrutura demonstrada nas Figuras 32 e 33.

Por fim, os serviços dos módulos precisam ser registrados no *framework* para funcionarem corretamente. Esse procedimento pode ser feito de duas maneiras. A primeira maneira, utiliza o modo nativo da especificação OSGi, o serviço é registrado dentro do método **start()** da interface **BundleActivator.java** que é implementada por todos os módulos. O registro é realizado no momento em que o **Módulo Log Core** é instalado, pois o mesmo é quem contém a implementação do serviço. Porém como o mesmo implementa a interface do **Módulo Log API**, esse módulo já deve estar instalado. A Figura 34 demonstra o código necessário dentro do método **start()** para que o serviço seja registrado.

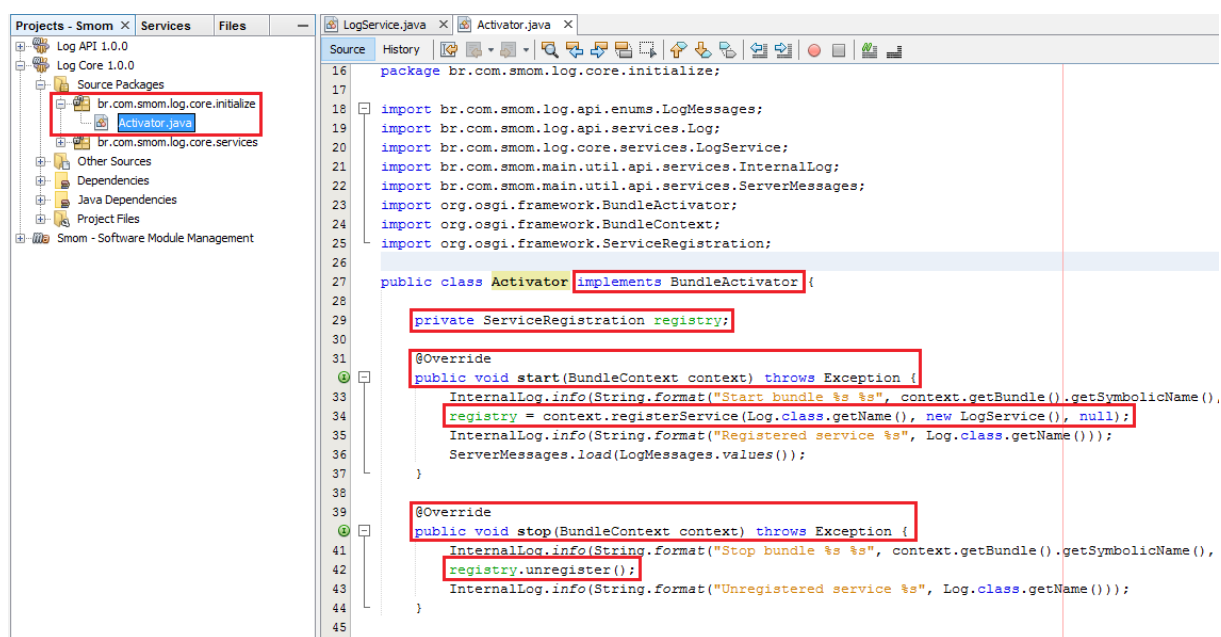


Figura 34 – Registro e remoção de serviços no *framework*. Fonte: Elaborado pelos autores

O código demonstrado na Figura 34 cria e remove o serviço do módulo no registro do *framework*. O objeto **context** do tipo **BundleContext** recebido no método **start()** é o responsável por realizar a criação do registro como é demonstrado na linha 34. A referência desse registro é armazenada no atributo **registry** do tipo **ServiceRegistration** que posteriormente pode ser usado para remover o registro desse serviço quando o método **stop()**

for invocando ao parar o módulo, como é demonstrado na linha 42.

A segunda maneira em que os serviços dos módulos podem ser registrados, é quando o módulo *core* é do tipo EJB. Nesse caso, quando o servidor de aplicação GlassFish, reconhece um novo módulo sendo instalado e encontra anotações `@Stateless` ou `@Singleton` da especificação EJB conforme demonstrado na Figura 35, o mesmo já realiza o registro de forma automática.

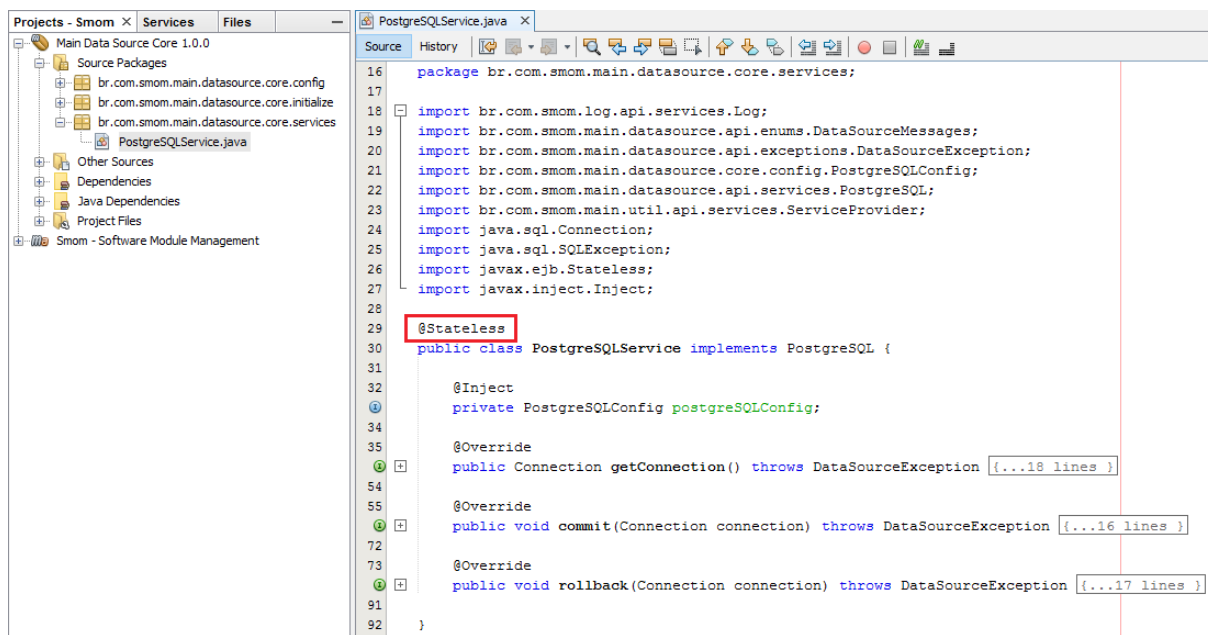


Figura 35 – Registro de serviços utilizando anotações da especificação EJB. Fonte: Elaborado pelos autores.

O serviço demonstrado na Figura 35 é do **Módulo Data Source Core** que implementa a interface **PostgreSQL.java** do **Módulo Data Source API**. Essa estrutura é a mesma do **Módulo de Log** apresentado anteriormente. Esse modo de registrar serviços utilizando anotações da especificação EJB é uma característica específica do servidor de aplicação GlassFish, ou seja, não funcionará ao utilizar outro servidor de aplicação.

Para realizar o gerenciamento dos módulos no sistema foi utilizado a ferramenta *Apache Felix Web Console Bundles* que oferece uma interface gráfica para trabalhar com os módulos desenvolvidos. Para instalar a mesma basta fazer o *download* no site²¹ da Apache Felix e colocá-la dentro da pasta **autostart** do servidor de aplicação GlassFish conforme demonstrado na Figura 36.

21 Link para download Apache Felix Web Console Bundles: <http://felix.apache.org/downloads.cgi>

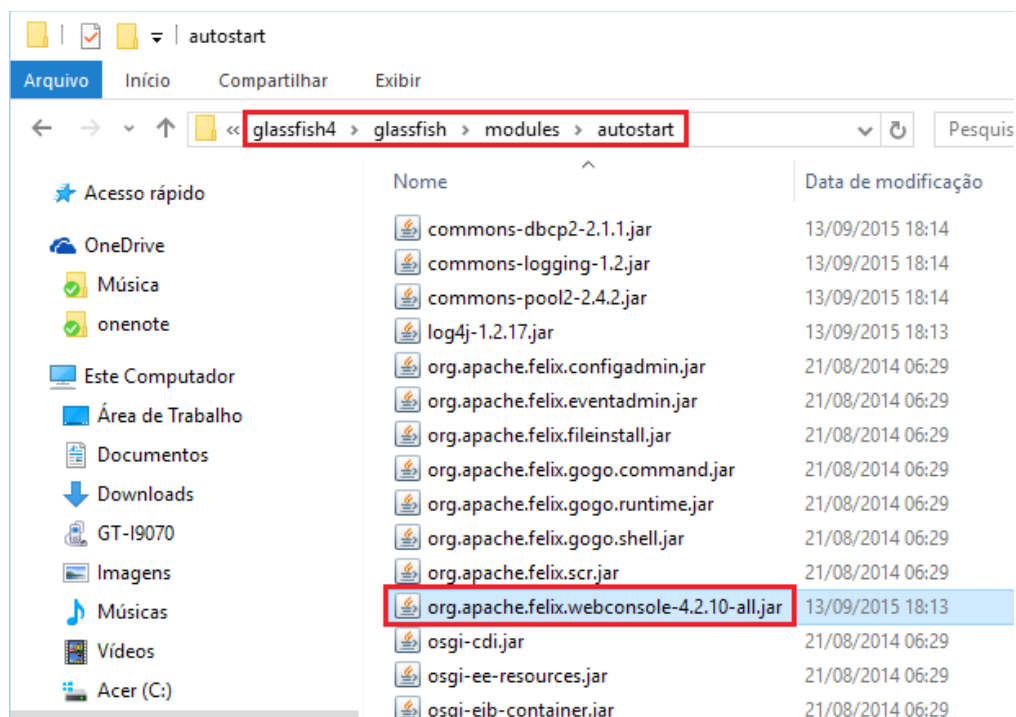
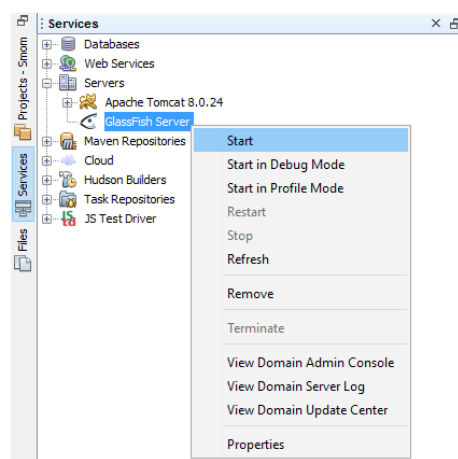


Figura 36 – Instalação da ferramenta Apache Felix Web Console Bundles. **Fonte:** Elaborado pelos autores.

Em seguida basta iniciar o servidor de aplicação conforme demonstra a Figura 37. O mesmo pode ser iniciado através da IDE NetBeans ou do comando **asadmin start-domain**.



```
D:\workspace\development\servers\windows\glassfish4>bin\asadmin.bat start-domain
Waiting for domain1 to start .....
Successfully started the domain : domain1
domain Location: D:\workspace\development\servers\windows\glassfish4\glassfish\domains\domain1
Log File: D:\workspace\development\servers\windows\glassfish4\glassfish\domains\domain1\logs\server.log
Admin Port: 4848
Command start-domain executed successfully.
```

Figura 37 – Inicialização do servidor de aplicação GlassFish. **Fonte:** Elaborado pelos autores.

Após iniciar o GlassFish, para ter acesso à ferramenta Web Console basta acessar através do navegador a url: `http://localhost:8080/osgi/system/console/bundles`. Será solicitado usuário e senha, ambas são “admin”. A Figura 38 mostra a página da ferramenta.

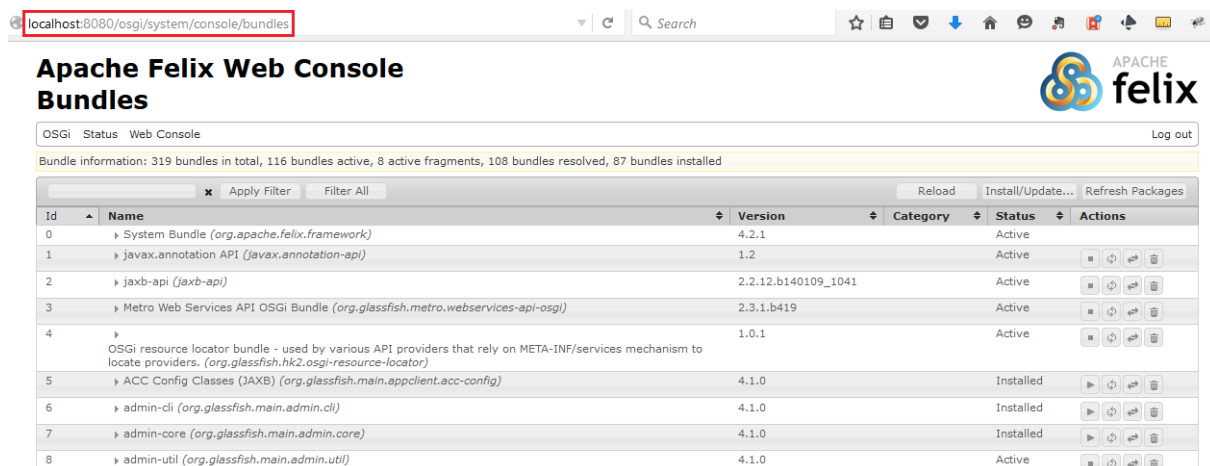


Figura 38 – Ferramenta Apache Felix Web Console Bundles. **Fonte:** Elaborado pelos autores.

Para instalar os módulos é necessário gerar o *build* dos módulos, para isso basta clicar com o botão direito sobre o projeto principal que está na IDE NetBeans e escolher a opção “Clean and Build”. Através dessa opção, a IDE constrói os projetos conforme suas configurações. Agora basta através da ferramenta Web Console clicar no botão **Install/Update** e escolher o módulo a ser instalado conforme a Figura 39.

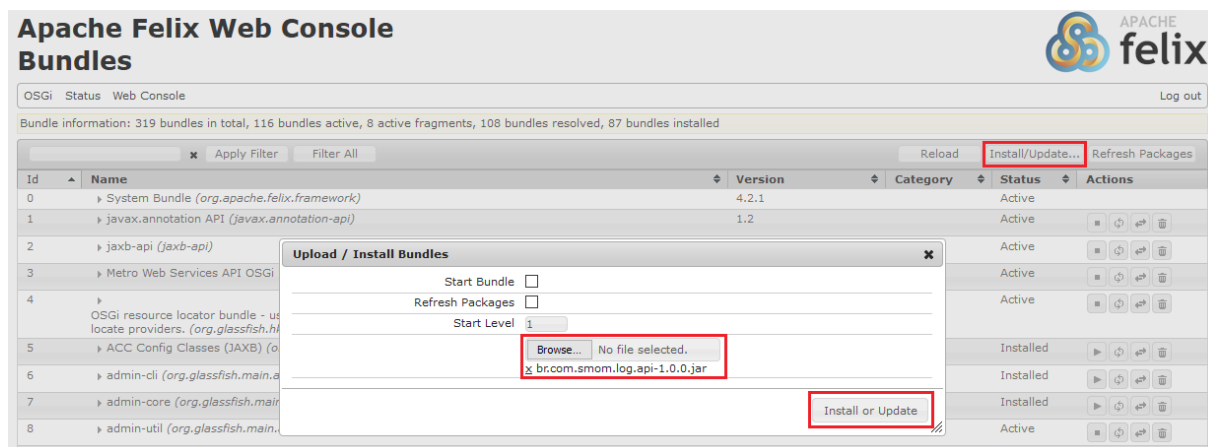


Figura 39 – Instalação de módulos através do Web Console. **Fonte:** Elaborado pelos autores.

Após realizar esse procedimento o módulo é instalado e fica disponível para ser gerenciado dentro do *framework*.

Todos esses procedimentos realizados no desenvolvimento foram necessários para a construção do *software* modularizado, os mesmos são de extrema importância para o funcionamento e comunicação dos módulos. Dessa forma, o *software* se torna desacoplado e flexível.

3.5 Resultados

Os resultados obtidos com a realização dos procedimentos se resumem de forma geral na estrutura do projeto e no funcionamento do *software*.

A Figura 40 demonstra como ficou a estrutura completa de todos os projetos criados dentro da IDE NetBeans. Através desta estrutura é visível a semelhança com a arquitetura do *software* demonstrada na Figura 12.

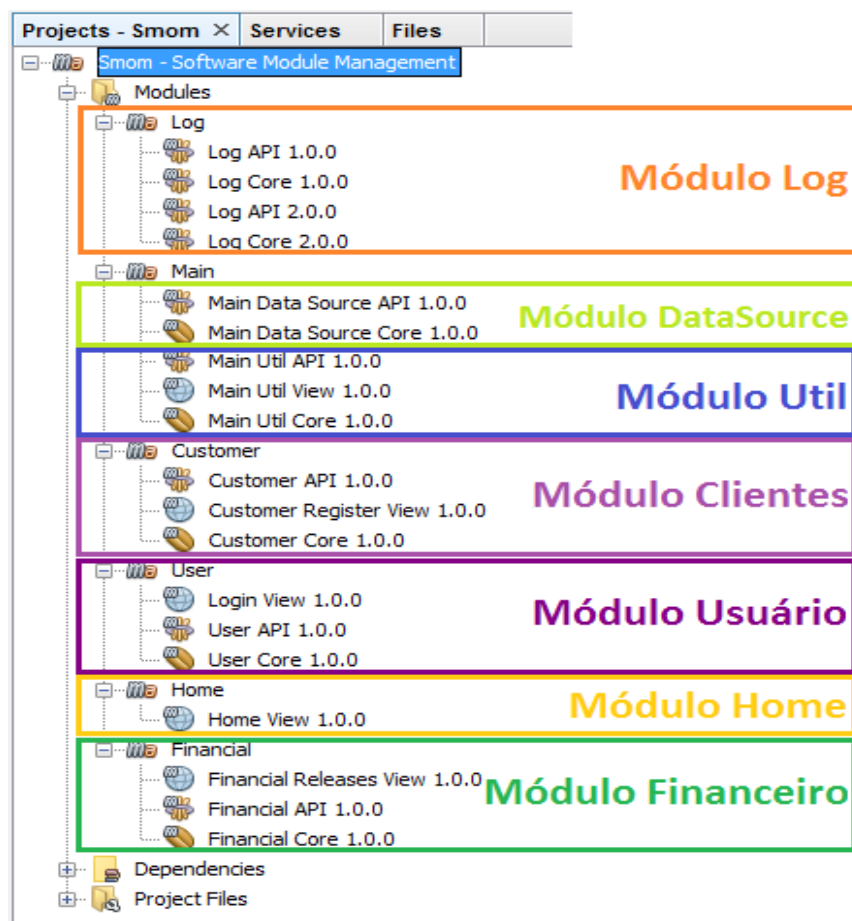


Figura 40 – Estrutura do projeto na IDE Netbeans. **Fonte:** Elaborado pelos autores.

Com essa estrutura pronta, é possível começar a utilizar os serviços que os módulos dispõem dentro do *software*. O *framework* utilizado no desenvolvimento dispõe através da classe **Framework.java** uma forma de obter as instâncias de serviços dos módulos. Sendo assim foi criado uma classe responsável por gerenciar essa obtenção dos serviços, facilitando assim a utilização dos outros módulos. A Figura 41 demonstra, depois dos módulos já estruturados, como o serviço de um módulo é utilizado. Nesse exemplo, é mostrado o **Módulo Data Source Core** utilizando um serviço do **Módulo Log** para gravar um *log* no sistema após obter a conexão com o banco de dados.

```
18 import br.com.smom.log.api.services.Log;
19 import br.com.smom.main.datasource.api.enums.DataSourceMessages;
20 import br.com.smom.main.datasource.api.exceptions.DataSourceException;
21 import br.com.smom.main.datasource.core.config.PostgreSQLConfig;
22 import br.com.smom.main.datasource.api.services.PostgreSQL;
23 import br.com.smom.main.util.api.services.ServiceProvider;
24 import java.sql.Connection;
25 import java.sql.SQLException;
26 import javax.ejb.Stateless;
27 import javax.inject.Inject;
28
29 @Stateless
30 public class PostgreSQLService implements PostgreSQL {
31
32     @Inject
33     private PostgreSQLConfig postgresSQLConfig;
34
35     @Override
36     public Connection getConnection() throws DataSourceException {
37
38         Log logService = (Log) ServiceProvider.getBundleService(Log.class);
39         Connection connection;
40
41         try {
42             connection = postgresSQLConfig.getConnection();
43             if (logService != null) {
44                 logService.info(DataSourceMessages.INFO_GET_CONNECTION_POSTGRES.getMessage());
45             }
46             return connection;
47         }
```

Figura 41 – Obtendo serviço do Módulo Log. Fonte: Elaborado pelos autores.

Pode-se observar que na linha 38 através do método **ServiceProvider.getBundleService()** é obtido o serviço do **Módulo Log**. Ao solicitar esse serviço, é necessário informar como parâmetro “**Log.class**” que é a interface **Log.java** e está no **Módulo Log API**, porém o *framework* retorna de fato uma instância da implementação que está no **Módulo Log Core**. Com a instância do serviço referenciada na variável **logService**, basta usá-la conforme o código da linha 43.

Através disso percebe-se que não há uma dependência direta entre a interface que

disponibiliza o serviço e o código que a implementa, ficando a cargo do *framework* gerenciar os serviços disponibilizados e consumidos, além de suas implementações.

Com isso verificou-se que o *software* está coeso e ao mesmo tempo desacoplado, pois é possível gerenciar os módulos sem interferir no funcionamento dos outros, com exceção é claro dos módulos principais do sistema, em que os mesmos são a base para o funcionamento do *software*.

Por fim, obteve-se o sistema funcionando. A Figura 42 mostra o *software* em funcionamento na sua tela principal.

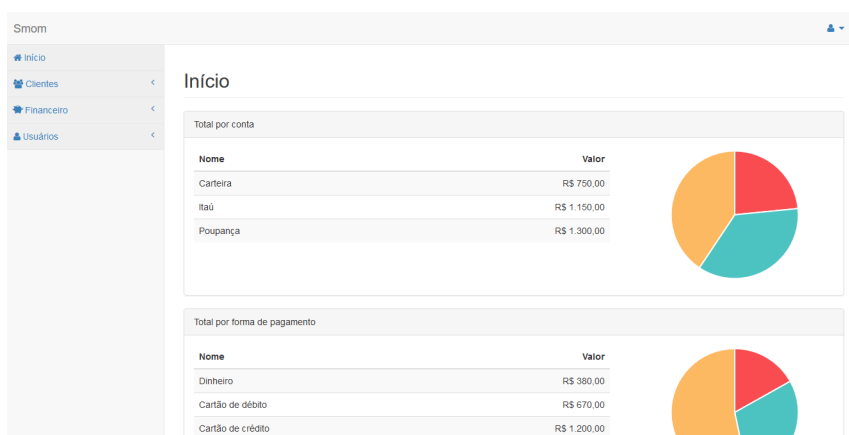


Figura 42 – Tela principal do *software*. **Fonte:** Elaborado pelos autores.

A Figura 43 demonstra os módulos de Clientes, Financeiro e Usuários ativos no menu do sistema.

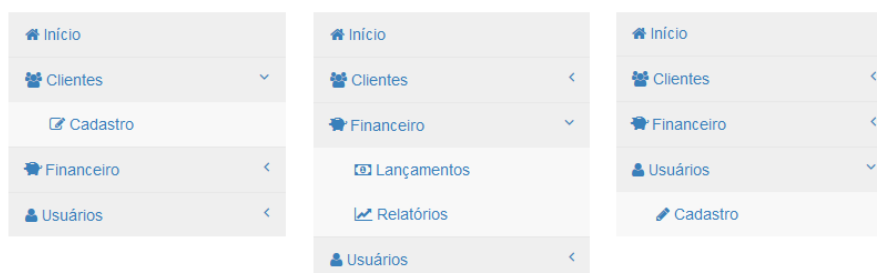


Figura 43 – Módulos ativos no menu do sistema. **Fonte:** Elaborado pelos autores.

Vale salientar que todas as telas do sistema foram criadas utilizando as tecnologias HTML, JavaScript e CSS com os *frameworks* Bootstrap e Angular JS, que tornaram o desenvolvimento mais produtivo.

4 DISCUSSÃO DE RESULTADOS

Neste capítulo são apresentados e discutidos os resultados obtidos conforme a pesquisa realizada sobre modularização de *softwares* utilizando a tecnologia OSGi, com o objetivo de destacar suas características e os pontos principais.

O objetivo da pesquisa foi demonstrar um modelo de desenvolvimento modular utilizando a tecnologia OSGi, a qual fornece suporte para o mesmo. Dessa maneira decidiu-se desenvolver uma aplicação simples que exemplifique a utilização dessa tecnologia.

Em uma visão geral, a aplicação desenvolvida está composta pelos módulos de usuário, clientes, financeiro, *data source* e log. Porém, esses módulos por sua vez, são compostos por módulos menores, que proporcionam flexibilidade e desacoplamento dos mesmos no *software*.

Conforme afirma Knoernschild (2012), uma aplicação modular é aquela que seus módulos podem ser instalados, parados, reiniciados e desinstalados sem interromper o restante da aplicação, além de serem reutilizáveis, combináveis e oferecerem uma interface clara.

Com base nessa teoria e utilizando a tecnologia OSGi que oferece suporte a essas características, desenvolveu-se uma aplicação que permite que seus módulos possam ser instalados, desinstalados, parados, atualizados sem interromper o restante da aplicação. Os mesmos também podem ser reutilizáveis e combinados com outros módulos. É importante destacar que existem módulos que não devem ser parados ou desinstalados por serem partes principais do sistema.

Com exceção dos módulos principais, que são denominados como *Main Util API*, *Main Web Resources View*, e todos os outros módulos do tipo API, os demais módulos podem ser desinstalados, instalados, reiniciados e parados sem comprometer o restante do sistema, apenas parando de executar suas responsabilidades específicas.

A Figura 44 demonstra através da ferramenta *Apache Felix Web Console Bundles*, que possibilita o gerenciamento dos módulos, o módulo de clientes instalado. O mesmo é composto por três módulos nomeados de *Customer API*, *Customer Core* e *Customer Register View*.

Apache Felix Web Console Bundles



OSGi Status Web Console

Log out

Bundle information: 313 bundles in total, 109 bundles active, 8 active fragments, 100 bundles resolved, 96 bundles installed

Apply Filter

Filter All

Reload

Install/Update...

Refresh Packages

<div>Id</div>	<div>Name</div>	<div>Version</div>	<div>Category</div>	<div>Status</div>	<div>Actions</div>
327	Customer Register View (br.com.smom.customer.register.view)	1.0.0		Active	<div><div></div><div></div><div></div><div></div></div>
326	Customer Core (br.com.smom.customer.core-v1)	1.0.0		Active	<div><div></div><div></div><div></div><div></div></div>
325	Customer API (br.com.smom.customer.api-v1)	1.0.0		Active	<div><div></div><div></div><div></div><div></div></div>
324	Main Web Resources View (br.com.smom.main.webresources.view)	1.0.0		Active	<div><div></div><div></div><div></div><div></div></div>
323	Main Util API (br.com.smom.main.util.api-v1)	1.0.0		Active	<div><div></div><div></div><div></div><div></div></div>
317	Apache Log4j (log4j)	1.2.17		Active	<div><div></div><div></div><div></div><div></div></div>
310	Apache Commons Pool (org.apache.commons.pool2)	2.4.2		Active	<div><div></div><div></div><div></div><div></div></div>
309	Apache Commons Logging (org.apache.commons.logging)	1.2.0		Active	<div><div></div><div></div><div></div><div></div></div>
308	Apache Commons DBCP (org.apache.commons.dbcp2)	2.1.1		Active	<div><div></div><div></div><div></div><div></div></div>
303	PostgreSQL JDBC Driver JDBC41 (org.postgresql.jdbc41)	9.4.0.build-1201		Active	<div><div></div><div></div><div></div><div></div></div>

Figura 44 – Módulo API, Core e View do Módulo Clientes instalados e ativos. **Fonte:** Elaborado pelos autores.

A Figura 45 demonstra que o módulo de clientes está disponível no menu do sistema. Dos três módulos que compõe o módulo de clientes, é possível parar ou desinstalar o módulo *Customer Register View* e *Customer Core* sem afetar o restante do sistema.

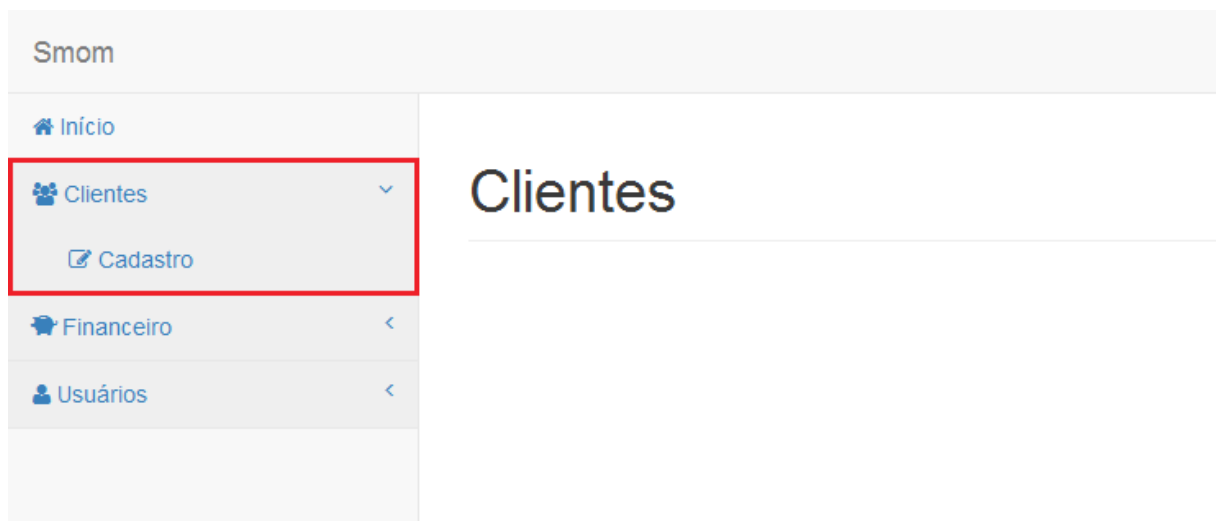


Figura 45 – Módulo de cadastro de clientes ativo no sistema. **Fonte:** Elaborado pelos autores.

A Figura 46 demonstra através da ferramenta que gerencia os *bundles* esses dois módulos parados.

Apache Felix Web Console Bundles



































OSGi Status Web Console					Log out
Bundle information: 313 bundles in total, 109 bundles active, 8 active fragments, 100 bundles resolved, 96 bundles installed					
<input type="text"/> Apply Filter Filter All			Reload	Install/Update...	Refresh Packages
ID	Name	Version	Category	Status	Actions
327	Customer Register View (<i>br.com.smom.customer.register.view</i>)	1.0.0		Resolved	   
326	Customer Core (<i>br.com.smom.customer.core-v1</i>)	1.0.0		Resolved	   
325	Customer API (<i>br.com.smom.customer.api-v1</i>)	1.0.0		Active	   
324	Main Web Resources View (<i>br.com.smom.main.webresources.view</i>)	1.0.0		Active	   
323	Main Util API (<i>br.com.smom.main.util.api-v1</i>)	1.0.0		Active	   
317	Apache Log4j (<i>log4j</i>)	1.2.17		Active	   
310	Apache Commons Pool (<i>org.apache.commons.pool2</i>)	2.4.2		Active	   
309	Apache Commons Logging (<i>org.apache.commons.logging</i>)	1.2.0		Active	   
308	Apache Commons DBCP (<i>org.apache.commons.dbcp2</i>)	2.1.1		Active	
303	PostgreSQL JDBC Driver JDBC41 (<i>org.postgresql.jdbc41</i>)	9.4.0.build-1201		Active	

Figura 46 – Módulos Core e View do Módulo Clientes parados. Fonte: Elaborado pelos autores.

A Figura 47 demonstra o *software* ainda em funcionamento porém com o módulo de clientes indisponível para acesso.

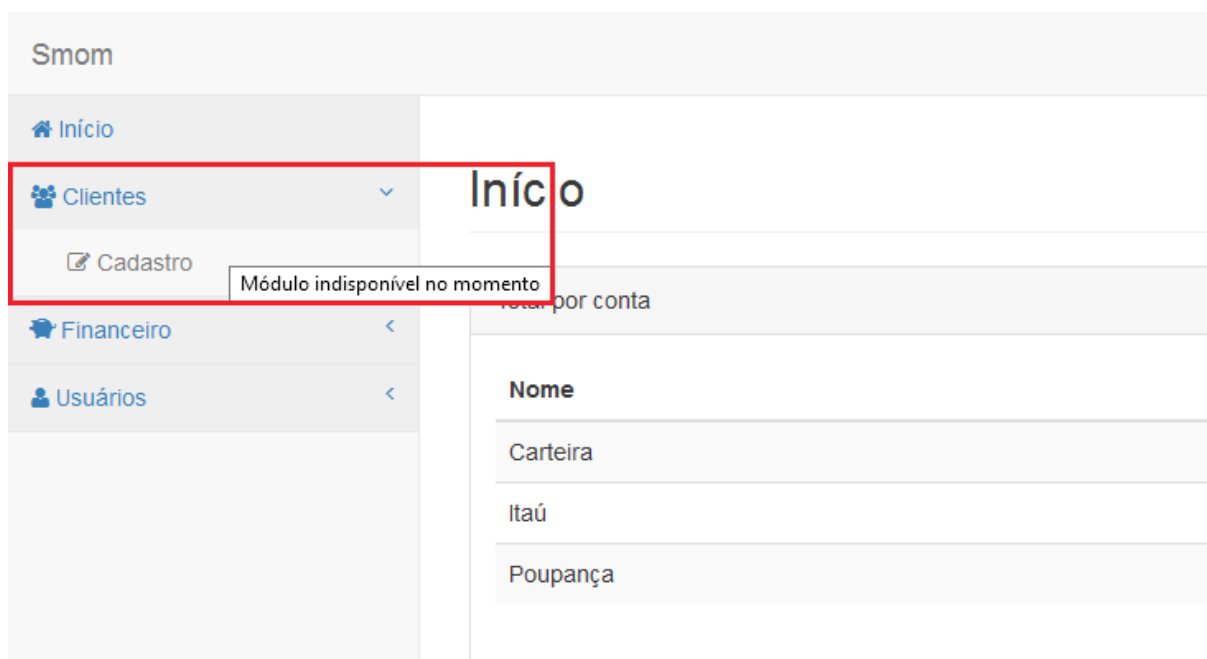


Figura 47 – Módulo de cadastro de clientes parado no sistema. Fonte: Elaborado pelos autores.





















Nesse exemplo os módulos *Customer Core* e *Customer Register View* foram parados e com isso o módulo de cadastro de cliente fica desativado no menu do sistema conforme demonstrado na Figura 47. Porém, se somente o módulo *Customer Register View* for parado o módulo de cadastro de clientes já fica desativado, com isso as funcionalidades do módulo *Customer Core* continuam funcionando. A Figura 48 mostra o que acontece ao acessar a página do cadastro de clientes com seu respectivo módulo parado.

Apache Felix Web Console Bundles



OSGi Status Web Console Log out

Bundle information: 314 bundles in total, 113 bundles active, 8 active fragments, 105 bundles resolved, 88 bundles installed

Id	Name	Version	Category	Status	Actions
327	Customer Register View (<i>br.com.smom.customer.register.view</i>)	1.0.0		Resolved	   
326	Customer Core (<i>br.com.smom.customer.core-v1</i>)	1.0.0		Active	   
325	Customer API (<i>br.com.smom.customer.api-v1</i>)	1.0.0		Active	   
324	Main Web Resources View (<i>br.com.smom.main.webresources.view</i>)	1.0.0		Active	   
323	Main Util API (<i>br.com.smom.main.util.api-v1</i>)	1.0.0		Active	   

localhost:8080/modules/customer/register/ Search

 Módulo indisponível no momento

Módulo indisponível no momento, contate o administrador do sistema ou tente mais tarde.

Figura 48 – Módulo View do cadastro de clientes parado no sistema. Fonte: Elaborado pelos autores.

Utilizando a tecnologia OSGi e estruturando os módulos através de uma arquitetura que forneça flexibilidade e o desacoplamento dos módulos, confirma-se então a teoria de Knoernschild, que um sistema modular é aquele onde é possível gerenciar seus módulos sem comprometer o restante da aplicação.

Umas das vantagens da modularização é a reutilização das funcionalidades que uma vez implementadas, podem ser reutilizadas em outros sistemas ou partes internas do mesmo sistema e, para se conseguir fazer isso, foi necessário definir bem a estrutura do sistema, organizando as funcionalidades e responsabilidades de cada módulo, de forma em que sua arquitetura permitisse uma alta reutilização de funcionalidades. Porém, essa organização da arquitetura exige que nos momentos iniciais do desenvolvimento do *software* seja criado uma granularidade adequada para o mesmo, se atendendo para a produção de módulos coesos e desacoplados o máximo possível.

Conforme Knoernschild (2012) a granularidade é formada pelo tamanho ou complexidade dos módulos, quanto menor a granularidade do *software*, menores são os módulos, possuindo menos inteligência de negócio neles, isso facilita a reutilização dos módulos, porém dificulta o uso deles, pois o sistema ao todo terá um grande número de módulos, tornando-se mais complicado e difícil de usar e expandir o sistema do que reutilizar.

Em contrapartida temos a alta granularidade, onde um módulo é mais grande e possui

mais inteligência de negócio, usar este módulo irá ser fácil, pois para executar uma rotina é necessário realizar apenas uma chamada de serviço de determinado módulo, porém no momento em que alguma funcionalidade que está presente dentro de algum módulo tiver de ser reutilizada em outro local da aplicação, não vai ser possível, pois essa funcionalidade está encapsulada junto às outras dentro do módulo, impedindo assim de reutilizá-la (KNOERNSCHILD, 2012).

Com base na teoria citada pelo autor Knoernschild, foram desenvolvidos vários protótipos até que encontrou-se a granularidade ideal para podermos desenvolver um *software* que exemplifica o modelo de desenvolvimento modular. Conforme mostra a Figura 49, essa foi a granularidade encontrada para o *software*. É extremamente importante salientar que cada projeto vai possuir uma granularidade própria, ou seja, dependendo do *software* e do objetivo a ser atingido, essa granularidade estará se ajustando. Com isso essa granularidade encontrada pode ser profundamente alterada dependendo do projeto a ser desenvolvido utilizando OSGi.

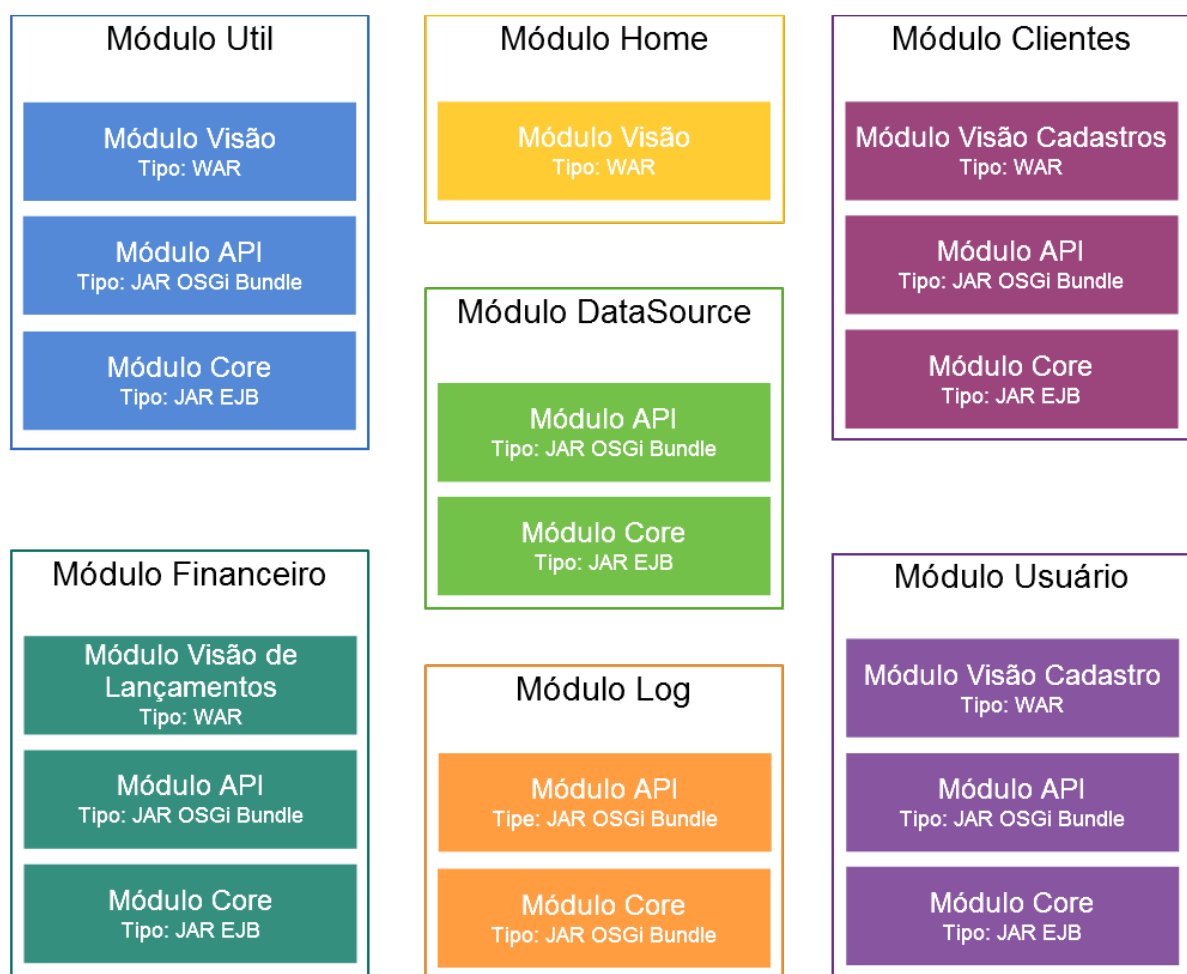


Figura 49 – Diagrama representando a granularidade do *software*. **Fonte:** Elaborado pelos autores.

Nesta granularidade dividiu-se cada um dos módulos em outros menores, em que cada módulo principal é composto por outros três módulos menores:

- A parte das interfaces dos serviços disponibilizadas aos outros módulos, ou seja, um módulo que contém a API;
- A parte da implementação dessas interfaces, nessa parte se teria as classes e códigos desenvolvidos em Java que implementam as interfaces e teriam consigo a inteligência e regras de negócio da aplicação, essa parte é o módulo *Core* da aplicação;
- A parte da visualização e interação com o usuário, o *front-end* deste módulo. Nesta parte estaria toda o código *web* que realiza a interação com usuário.

Alguns módulos como o de *Log* e *Data Source* não são compostos por essa estrutura, os mesmos não possuem a parte de interação com o usuário, pois apenas servem como funcionalidades utilizadas pelos outros módulos.

A granularidade resultante do desenvolvimento deste *software*, possibilita a reutilização de funcionalidades, tanto por outros módulos como por outros *softwares*.

Além disso, com essa divisão entre as funcionalidades e códigos do sistema, foi possível obter uma alta coesão e desacoplamento, uma vez que com essa granularidade cada módulo executa apenas a sua funcionalidade, além de não depender diretamente da implementação de outros módulos.

Com base nas funcionalidades dos módulos do sistema, caso as funcionalidades fossem divididas para cada módulo, seria obtida uma alta granularidade, assim o reaproveitamento de alguma funcionalidade presente dentro do módulo de financeiro, por exemplo, estaria encapsulada, pois para reaproveitar em outra parte do sistema seria necessário levar junto o restante de códigos e funcionalidades do módulo, o que não é aceitável, pois teríamos no sistema código duplicado e que afetaria a coesão da implementação.

Outro ponto a destacar é o controle de versionamento que a tecnologia OSGi oferece, na qual é possível desenvolver dois módulos iguais porém com versões diferentes que podem ser utilizados por diferentes módulos do sistema.

O versionamento pode ser interessante em casos que é necessário lançar uma nova versão com alguma funcionalidade que somente uma parte de todo o *software* utilizará, ou

ainda uma nova funcionalidade que esteja em testes. Para exemplificar uma situação criou-se a versão 2.0.0 do **Módulo Log**. Em sua versão 1.0.0 este módulo utiliza o *framework* log4j para escrever os *logs* do sistema, enquanto na versão 2.0.0 o módulo passa a registrar os *logs* utilizando a API nativa do Java. A Figura 50 demonstra o **Módulo Log** na versão 1.0.0 e 2.0.0 instalados e ativos.

Apache Felix Web Console Bundles



OSGi Status Web Console Log out

Bundle information: 319 bundles in total, 116 bundles active, 8 active fragments, 108 bundles resolved, 87 bundles installed

Apply Filter Filter All

Reload Install/Update... Refresh Packages

<div><div>Id</div></div>	<div><div>Name</div></div>	<div><div>Version</div></div>	<div><div>Category</div></div>	<div><div>Status</div></div>	<div><div>Actions</div></div>
320	Log API (br.com.smom.log.api-v1)	1.0.0		Active	<div><div></div><div></div><div></div><div></div></div>
324	Log Core (br.com.smom.log.core-v1)	1.0.0		Active	<div><div></div><div></div><div></div><div></div></div>
321	Log API (br.com.smom.log.api-v2)	2.0.0		Active	<div><div></div><div></div><div></div><div></div></div>
325	Log Core (br.com.smom.log.core-v2)	2.0.0		Active	<div><div></div><div></div><div></div><div></div></div>

Apply Filter Filter All

Reload Install/Update... Refresh Packages

Bundle information: 319 bundles in total, 116 bundles active, 8 active fragments, 108 bundles resolved, 87 bundles installed

Figura 50 – Módulo Log instalado com a versão 1.0.0 e 2.0.0. Fonte: Elaborado pelos autores.

A Figura 51 demonstra o **Módulo DataSource** utilizando a versão 2.0.0 do Módulo de Log. No trecho do código não é possível perceber qual a versão que está sendo utilizada, pois o serviço do **Módulo Log** ainda é o mesmo. O que muda entre as versões é somente a implementação. O *framework* gerencia esse versionamento através do arquivo MANIFEST.MF de cada módulo.

```
Log logService = (Log) ServiceProvider.getBundleService(Log.class);
Connection connection;

try {
    connection = postgresSQLConfig.getConnection();
    if (logService != null) {
        logService.info(DataSourceMessages.INFO_GET_CONNECTION_POSTGRES.getMessage());
    }
    return connection;
} catch (SQLException e) {
    if (logService != null) {
        logService.error(DataSourceMessages.ERROR_GET_CONNECTION_POSTGRES.getMessage(), e);
    }
    throw new DataSourceException(DataSourceMessages.ERROR_GET_CONNECTION_POSTGRES, e);
}
```

```
Import-Package: br.com.smom.log.api.services;version="[2.0,3]",br.com.smom.main.datasource.api.enums;version="[1.0,2)",br.com.smom.main.datasource.api.exceptions;version="[1.0,2)",br.com.smom.main.datasource.api.services;version="[1.0,2)",br.com.smom.main.util.api.enums;version="[1.0,2)",br.com.smom.main.util.api.services;version="[1.0,2)",javax.ejb,javax.enterprise.context,javax.inject,org.apache.commons.dbcp2;version="[2.1,3)",org.osgi.framework;version="[1.6,2)"
```

Figura 51 – Trecho de código e dependências demonstrando o versionamento. Fonte: Elaborado pelos autores.

Através dos resultados discutidos, pode-se demonstrar algumas das vantagens do desenvolvimento modular utilizando a tecnologia OSGi. As mesmas seriam de grande utilidade para *software* que contém requisitos como desacoplamento, alta coesão, reutilização de funcionalidades, versionamento e ainda a manutenção e expansão do sistema sem comprometer totalmente o funcionamento do sistema.

A utilização do OSGi exige maior tempo e esforço de aprendizagem, já que o mesmo impõe conceitos sobre modularização, utilização de interfaces e serviços.

Esses fatores podem ser confirmados conforme diz Gama (2008), que “existe uma curva de aprendizado que não vale a pena e nem faz sentido se você está desenvolvendo aplicações que não precisam das vantagens do OSGi”.

REFERÊNCIAS

- APACHE. **OSGi Frequently Asked Questions**. 2015a. Disponível em <http://felix.apache.org/documentation/tutorials-examples-and-presentations/apache-felix-osgi-faq.html>. Acesso em 16 de junho, 2015.
- APACHE. **What is maven**. 2015b. Disponível em <http://maven.apache.org/what-is-maven.html>. Acesso em 16 de junho, 2015.
- APPOLINÁRIO, Fábio. **Dicionário de metodologia: um guia para a produção do conhecimento científico**. São Paulo: Atlas, 2004.
- BORBA, Paulo. **Aspectos de Modularização**. 2015. Disponível em <http://www.di.ufpe.br/~java/graduacao961/aulas/aula4/aula4.html>. Acesso em 21 de junho, 2015.
- BARTLETT, Neil. **OSGi In Practice**. 2009. Disponível em http://njbartlett.name/files/osgibook_preview_20091217.pdf. Acesso em 09 de maio, 2015.
- BEZERRA, Eduardo. **Princípios de Análise e Projeto de Sistemas com UML**. Rio de Janeiro: Campus, 2002.
- BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML: Guia do Usuário**. Rio de Janeiro: Campus, 2000.
- BOSSCHAERT, David. **OSGi in Depth**. Shelter Island: Manning Publications Co, 2012
- CAELUM. **Apostila Java para Desenvolvimento Web**. 2015a. Disponível em <https://www.caelum.com.br/apostila-java-web/>. Acesso em 08 de agosto, 2015.
- CAELUM. **Apostila Desenvolvimento Web com HTML, CSS e JavaScript**. 2015b. Disponível em <https://www.caelum.com.br/apostila-html-css-javascript/javascript-e-interatividade-na-web/>. Acesso em 08 de março, 2015.
- CAELUM. **Apostila Java e Orientação a Objetos**. 2015c. Disponível em <http://www.caelum.com.br/apostila-java-orientacao-objetos/>. Acesso em 08 de março, 2015.
- CLARO, Daniela Barreiro; SOBRAL, João Bosco Manguiera. **Programação em Java**. Santa Catarina: Cengage Learning Pearson Education, 2008.
- COOK, Stuart; SELTZER, Claire; WRIGHTSMAN, Lawrence Samuel. **Métodos de pesquisa nas relações sociais**. São Paulo: EPU, 1987.
- COSTA, Gabriel. **O que é bootstrap?** 2014. Disponível em <http://www.tutorialwebdesign.com.br/o-que-e-bootstrap/>. Acesso em 09 de agosto, 2015.

DEITEL, Harvey Matt; DEITEL, Paul John. **Java How to Program**. 8. ed. Edson Furmankiewicz. São Paulo: Pearson Prentice Hall, 2010.

DEVMEDIA. **Novidades do GlassFish 3.1**. 2011. Disponível em: <http://www.devmedia.com.br/novidades-do-glassfish-3-1-artigo-java-magazine-91/21124>. Acesso em 19 de junho, 2015.

FERNANDES, Leonardo. **OSGi e os benefícios de uma Arquitetura Modular**. 37.ed. 2009. p. 27-35.

GAMA, Kiev. **Uma visão geral sobre a plataforma OSGi**. 2008. Disponível em <https://kievgama.wordpress.com/2008/11/24/um-pouco-de-osgi-em-portugues/>. Acesso em 09 de março, 2015.

GIL, Antônio Carlos. **Como elaborar projetos de pesquisa**. São Paulo: Atlas, 2002.

GONÇALVES, Antonio. **Beginning Java EE 6 Platform with GlassFish 3**. Nova York: Springer Science+Business Media, LCC. 2010.

JERSEY. **Jersey: RESTful Web Services in Java**. 2015. Disponível em <https://jersey.java.net/>. Acesso em 10 de agosto, 2015.

KIOSKEA. **Condução de reunião**. 2014. Disponível em <http://pt.kioskea.net/contents/579-conducao-de-reuniao>. Acesso em 16 de abril, 2015.

KNOERNSCHILD, Kirk. **Java Application Architecture: Modularity Patterns with Examples Using OSGi**. Crawfordsville: Pearson Education, 2012.

LUCENA, Fábio Nogueira de. **Introdução ao Equinox**. 2010. Disponível em <https://code.google.com/p/exemplos/wiki/equinox>. Acesso em 09 de março, 2015.

MALCHER, Marcelo Andrade da Gama. **OSGi Distribuído: deployment local e execução remota**. Monografia de Seminários de Sistemas Distribuídos. Pontifícia Universidade Católica do Rio de Janeiro, 2008.

MARIE, Victor. **Bootstrapt e formulários HTML5**. 2015. Disponível em <http://www.caelum.com.br/apostila-html-css-javascript/bootstrap-e-formularios-html5/>. Acesso em 09 de agosto, 2015.

MAUJOR. **Site sobre CSS e Padrões Web: Por que CSS?**. 2015. Disponível em <http://www.maujor.com/index.php>. Acesso em 08 de março, 2015.

MAYWORM, Marcelo. **OSGi Distribuída: Uma Visão Geral**. 42.ed. 2010. p. 60-67.

MILANI, André. **PostgreSQL: Guia do Programador**. São Paulo: Novatec Editora, 2008.

Mozilla Developer Network. **CSS**. 2015. Disponível em: <https://developer.mozilla.org/pt->

BR/docs/Web/CSS. Acesso em 08 de março, 2015.

Mozilla Developer Network. **HTML**. 2014. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTML>. Acesso em 07 de março, 2015.

Mozilla Developer Network. **JavaScript**. 2015. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em 08 de março, 2015.

Mozilla Developer Network. **JavaScript language resources**. 2014. Disponível em https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources. Acesso em 08 de março, 2015.

NETBEANS. **Trabalhando com Injeção e Qualificadores no CDI**. 2015a. Disponível em https://netbeans.org/kb/docs/javaee/cdi-inject_pt_BR.html. Acesso em 09 de agosto, 2015.

NETBEANS. **Trilha do Aprendizado do Java EE e Java Web**. 2015b. Disponível em https://netbeans.org/kb/trails/java-ee_pt_BR.html. Acesso em 12 de agosto, 2015.

OLIVEIRA, Eric. **Aprenda AngularJS com estes 5 Exemplos Práticos**. 2013. Disponível em <http://javascriptbrasil.com/2013/10/23/aprenda-angularjs-com-estes-5-exemplos-praticos/>. Acesso em 09 de agosto, 2015.

OLIVEIRA, Jefferson Amorin de; WERLANG, Luciane Pires. **Aplicação da Modularização na Arquitetura e Desenvolvimento de um Componente de Pesquisa Baseado em Java**. Trabalho de Conclusão de Curso em Ciência da Computação na Universidade do Sul de Santa Catarina. Palhoça, 2006.

OSGI ALLIANCE. **OSGi**. 2015. Disponível em <http://www.osgi.org/Main/HomePage>. Acesso em 08 de março, 2015.

ORACLE. **Difference between GlassFish Open Source and Commercial Editions**. 2011. Disponível em https://blogs.oracle.com/GlassFishForBusiness/entry/difference_between_glassfish_open_source. Acesso em 20 de junho, 2015.

ORACLE. **Enterprise JavaBeans Technology**. 2015. Disponível em <http://www.oracle.com/technetwork/java/javaee/ejb/index.html>. Acesso em 10 de agosto, 2015.

PRESSMAN, Roger S. **Engenharia de Software**. 1 ed. São Paulo: Makron Books, 1995.

SAUDATE, Alexandre. **REST: Construa API's inteligentes de maneira simples**. São Paulo: Casa do Código, 2013.

SANTOS FILHO, Walter dos. **Introdução ao Apache Maven**. Belo Horizonte: Eteg Tecnologia da Informação Ltda, 2008.

SANTOS, Wagner Roberto dos. **RESTful Web Services e a API JAX-RS**. 2009. Disponível

em <http://www.univale.com.br/unisite/mundo-j/artigos/35RESTful.pdf>. Acesso em 08 de agosto, 2015.

SCHMITZ, Daniel; LIRA, Douglas. **AngularJS na prática**. 2014. Disponível em samples.leanpub.com/livro-angularJS-sample.pdf. Acesso em 09 de agosto, 2015.

SILVA, Maurício Samy. **CSS3: Desenvolva aplicações web profissionais com uso dos poderosos recursos de estilização das CSS3**. São Paulo: Novatec Editora, 2012.

SILVA, Maurício Samy. **HTML5: A linguagem de marcação que revolucionou a web**. São Paulo: Novatec Editora, 2011.

SILVA, Maurício Samy. **JavaScript: Guia do Programador**. São Paulo: Novatec Editora, 2010.

SOUZA, Arthur Câmara; AMARAL, Hugo Richard; LIZARDO, Luis Eduardo O. **PostgreSQL: uma alternativa para sistemas gerenciadores de banco de dados de código aberto**. In: Anais do Congresso Nacional Universidade, EAD e Software Livre, 2012.

STERN, Eduardo Hoelz. **PostgreSQL - Introdução e Conceitos**. 2003. Disponível em <http://www.devmedia.com.br/artigo-sql-magazine-6-postgresql-introducao-e-conceitos/7185>. Acesso em 10 de agosto, 2015.

STAA, Arndt von. **Programação Modular: desenvolvendo programas complexos de forma organizada e segura**. Rio de Janeiro: Campus, 2000.

USP. **Fundamentos do projeto de software**. 2015. Disponível em <http://www.pcs.usp.br/~pcs722/98/Objetos/bases.html>. Acesso em 21 de junho, 2015.

VOGEL, Lars. **OSGi Modularity – Tutorial**. 2015. Disponível em <http://www.vogella.com/tutorials/OSGi/article.html>. Acesso em 18 de abril, 2015.

W3C. **About W3C**. 2015. Disponível em <http://www.w3.org/Consortium/>. Acesso em 07 de março, 2015.

W3C. **What is CSS?**. 2015. Disponível em <http://www.w3.org/Style/CSS/>. Acesso em 08 de março, 2015. COOK, Stuart; SELLTIZ, Claire; WRIGHTSMAN, Lawrence Samuel. **Métodos de pesquisa nas relações sociais**. São Paulo: EPU, 1987.