Universidade do Minho
Escola de Engenharia

Luis Fernando Rodrigues Loureiro dos Santos

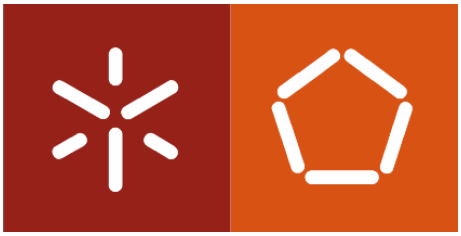
Modularidade em Java
O impacto do projeto Jigsaw

Julho 2011

Modularidade em Java
O impacto do projeto Jigsaw

Luis Santos

UMinho | 2011



Universidade do Minho
Escola de Engenharia

Luis Fernando Rodrigues Loureiro dos Santos

Modularidade em Java
O impacto do projeto Jigsaw

Tese de Mestrado em Informática

Trabalho realizado sob a orientação do
Professor Doutor António Manuel Nestor Ribeiro

Julho 2011

Universidade do Minho, 28/07/2011

Assinatura: 

Agradecimentos

Este foi um projeto longo, que exigiu esforço e que só foi conseguido porque várias pessoas ajudaram.

Agradeço desde logo ao orientador da dissertação, Professor Doutor António Manuel Nestor Ribeiro, todo o apoio fornecido, as ideias que apresentou e a forma como compreendeu a minha situação pessoal e profissional e me apoiou fornecendo indicações concisas que permitiram a obtenção dos resultados esperados.

Agradeço também a compreensão da empresa onde trabalho, que fez um esforço adicional, suprindo a minha ausência com outros colegas ou alterando agendas para que, através das várias ausências ao trabalho, me possibilitasse completar a dissertação.

Agradeço ainda a minha família, a minha esposa Marilia e o meu filho, Eduardo, que durante mais de um ano aceitaram as minhas ausências em família, para que me pudesse deslocar a Braga e para que pudesse completar a dissertação.

Por último, um agradecimento especial aos meus pais e especialmente à minha avó, a quem dedico este trabalho. Muito obrigado.

Resumo

A modularidade é um conceito importante na implementação de sistemas suportados por *software*. A linguagem Java é uma das linguagens utilizadas para implementar este tipo de sistemas.

Esta dissertação apresenta um estudo sobre os conceitos de modularidade que o projeto Jigsaw propõe para a linguagem Java, demonstrando como se comparam com o estado de arte de modularidade em ambientes de desenvolvimento Java, as melhorias para a linguagem Java e para os sistemas de software desenvolvidos em Java, nomeadamente sistemas baseados em servidores aplicacionais.

O projeto, através do conceito de modularidade proposto, introduz alterações importantes na linguagem e plataforma Java, na forma de desenvolvimento e distribuição de aplicações e esta dissertação pretende, através de análise e demonstração, mostrar a importância da metodologia apresentada e de que forma pode melhorar e substituir as várias metodologias de modularidade em Java atualmente existentes.

No âmbito desta dissertação, é apresentada uma aplicação informática, na forma de prova de conceito, desenvolvida utilizando a linguagem Java, que procura automatizar processos associados à aplicação da metodologia Jigsaw no desenvolvimento de aplicações.

As conclusões deste estudo permitem perceber que o Jigsaw apresenta melhorias significativas que devem ser incorporadas no Java mas, permitem também perceber a existência de limitações que devem ser corrigidas por forma a tornar o conceito mais abrangente para ser utilizado nos mais variados cenários, nomeadamente na implementação de aplicações complexas, como é o caso de servidores aplicacionais. A plataforma Java encontra-se numa fase de evolução

sensível, onde decisões que estão a ser tomadas pelas várias entidades que determinam o futuro da plataforma podem implicar o sucesso ou fracasso da plataforma, sendo o Jigsaw um ponto em aberto nesses processos de decisão.

Palavras-chave: modularidade em Java, metodologias de desenvolvimento, arquiteturas modulares, programação estruturada, Jigsaw

Abstract

Modularity is an important concept in the implementation of systems supported by software. Java is one of the languages used to implement such systems.

This thesis presents a study on the concepts of modularity presented in Jigsaw project for the Java language, showing how they compare with the state of the art of modularity in Java development environments, improvements to the Java language and the software systems developed in Java, in particular, on application servers.

The project, through the concept of modularity proposed, brings significant changes in the Java language and platform, to the form of developing and distributing applications and this thesis seeks, through analysis and demonstration, to show the importance of the methodology presented for the future of the platform and how it can improve and replace the various methodologies of modularity in Java that currently exists.

Under this thesis, we present a computer application (proof of concept), developed using the Java language, which seeks to automate processes associated with implementing the Jigsaw method in application development.

The findings of this study allow us to realize that Jigsaw has significant improvements that should be incorporated in Java but also allows to realize that there are limitations that should be corrected in order to make the concept more broadly to be used in various scenarios, including in the implementation of complex applications, such as the application servers. The Java platform is at a sensitive stage of development, where decisions are being taken by the various entities that determine the future of the platform that can lead to success or failure of the platform, with the Jigsaw as an open point in these decision processes.

Keywords: modularity in Java, development methodologies, modular architectures, structured programming, Jigsaw

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A modularidade é uma das metodologias utilizadas no desenho e desenvolvimento de aplicações informáticas. Das várias metodologias existentes, a modularidade é uma das mais frequentemente utilizadas.

A programação modular enquadra-se nas metodologias de programação estruturada. Estas têm evoluído ao longo dos anos, sendo assumido que tiveram as raízes na década de 60, quando a programação não estruturada começava a ter dificuldades em corresponder às necessidades cada vez mais exigentes de desenho e desenvolvimento de aplicações informáticas.

Estas metodologias possibilitaram a conceção de programas informáticos complexos, escritos de forma estruturada e com conceitos modulares, nomeadamente pelo aperfeiçoamento das metodologias baseadas em procedimentos, como é o caso das linguagens C, Pascal e Modula-2 e, representam, em última instância, as fundações para a programação orientada por objetos, uma das metodologias hoje em dia mais utilizadas.

As metodologias de desenvolvimento por objetos resultam, entre outros, de uma evolução das metodologias procedimentais e podem ser facilmente enquadradas na programação estruturada, incorporando muitos dos conceitos modulares já existentes e melhorando outras áreas que as linguagens de programação tradicionais (C, Pascal, Modula-2, …) não contemplavam.

Dentro do universo das linguagens de programação orientada por objetos,

surge o Java, atualmente uma das linguagens de programação mais populares no desenvolvimento de aplicações [TIOBE 10].

O Java teve uma forte aceitação [Gosling 05] desde que foi tornado público, em 1995, tanto no mundo académico como empresarial, sendo que a sua utilização abrange áreas tão distintas como sistemas de informação de larga escala, aplicações *Web*, caixas de receção de canais digitais e telemóveis [Horstmann 04].

A plataforma Java tem evoluído muito, incorporando em cada nova versão melhorias significativas e um maior número de classes e interfaces na *API*[1] – *Application Programming Interface* – da plataforma, tal como se pode verificar na tabela 1.1 [Horstmann 04].

Versão	Número de classes e interfaces
1.0	211
1.1	477
1.2	1524
1.3	1840
1.4	2723
5	3270

Tabela 1.1: Evolução do número de classes e interfaces na API da plataforma Java, versão J2SE

Atualmente, os projetos Java e o próprio ambiente Java podem apresentar níveis de elevada complexidade, com grandes interdependências entre si. A gestão destes projetos, nomeadamente das várias unidades de código e componentes que servem para implementar as funcionalidades desejadas, nos atos de desenvolvimento e acima de tudo na distribuição, é complexa [Turner 80] e propensa a erros, nomeadamente de inconsistências de versões entre blocos funcionais.

Apesar de todas as inovações presentes no Java, relativamente a outras

1 API – Application Programming Interface, traduzido para português, significa Interface de Programação de Aplicação.

linguagens e, do suporte de muitos dos conceitos que definem um processo estruturado de desenvolvimento de programas, o Java não é uma linguagem modular.

Dotar o Java de um sistema modular na própria definição da linguagem pode representar vantagens para a plataforma Java, permitindo tornar o Java mais completo e mais rico do ponto de vista de facilidades oferecidas a quem desenvolve software.

O projeto *Jigsaw*, nome de código dado ao projeto, promete resolver algumas das limitações da plataforma Java ao propor e implementar modularidade na linguagem Java e plataforma Java. Demonstrar as alterações propostas e as melhorias associadas à modularidade proposta permitirá contribuir para um maior entendimento dos avanços e, compreender o porquê da importância de aceitar uma alteração tão profunda no conceito de desenvolvimento e distribuição de sistemas de software Java.

1.1 Motivação

Hoje em dia, as aplicações tendem a ser cada vez mais complexas. São constituídas por muitos blocos de código, implementados de forma estruturada ou modular. São exemplos disso processadores de texto, folhas de cálculo, bases de dados e servidores aplicacionais.

Nos últimos 13 anos participei ativamente em projetos de desenvolvimento de software importantes, a nível empresarial, onde a linguagem de programação base foi o Java. Nestes projetos incluem-se sistemas de faturação, sistemas de gestão de relação com clientes e sistemas analíticos de dados. Estes sistemas, nomeadamente nas componentes de desenvolvimento, gestão de novas versões, correção de erros e incorporação de funcionalidades desenvolvidas por terceiros, só foram possíveis porque foram utilizadas metodologias estruturadas e modulares de desenvolvimento e distribuição das aplicações. Os principais benefícios identificados foram uma menor quantidade de recursos necessária, ciclos de desenvolvimento mais rápidos, maior estabilidade no processo de instalação das aplicações e, muito importante no mercado empresarial, poupanças significativas ao adotar desde o inicio nestes projetos uma

mentalidade modular.

Desta forma, percebo a importância de uma linguagem com o grau de aceitação e importância como é a linguagem Java incorporar elementos e características que a tornem mais eficaz e competitiva, tanto no sentido puramente científico/académico como de um ponto de vista de aplicabilidade empresarial.

A modularidade é uma estratégia que fortemente se enquadra e ajuda a cumprir os requisitos atrás enunciados.

O projeto Jigsaw propõe, ao introduzir conceitos de modularidade na linguagem Java e modularizar a plataforma Java, disponibilizar os mecanismos que permitam dotar a plataforma Java de modularidade e permitir também que as próprias aplicações desenvolvidas na plataforma Java possam ser desenvolvidas com metodologias modulares.

Desta forma, dada a importância que se espera ver mostrada, é importante demonstrar as mais valias da aplicação de conceitos reais de estratégias de modularidade no ambiente Java, com especial destaque para os propostos pelo projeto Jigsaw.

Esta dissertação pretende analisar o estado da arte de modularidade em Java, perceber de que forma se diferencia da proposta de modularidade do Jigsaw e se o Java, principalmente no que diz respeito ao desenvolvimento de aplicações, pode beneficiar com a modularidade Jigsaw.

1.2 Objetivos

Pretende-se com esta dissertação estudar em profundidade o conceito de modularidade proposto no projeto Jigsaw e demonstrar de que forma pode alterar a forma de desenvolvimento em Java.

Esta dissertação define como objetivos:

- Analisar metodologias de modularidade em Java, algumas das quais estão

no grupo das mais comuns na plataforma Java.

- Estudar o conceito de modularidade proposto pelo projeto Jigsaw para a plataforma Java.

- Apresentar as principais características que diferenciam o novo conceito proposto relativamente às metodologias de modularidade comummente utilizadas na linguagem Java.

- Identificar a importância para a plataforma Java e para a distribuição de aplicações Java da aceitação da especificação de modularidade Java representada pelo projeto Jigsaw.

- Identificar os efeitos resultantes da adoção desta nova metodologia no desenvolvimento Java, nos mais diversos ambientes, incluindo em ambientes de servidores aplicacionais, que pela sua complexidade, permitem uma avaliação mais abrangente.

- Apresentar uma aplicação informática (protótipo de validação de conceito) que, em circunstâncias perfeitamente definidas, ajude o programador a efetuar a correta divisão por módulos de um projeto de software e a fazer a gestão do ambiente de módulos segundo a metodologia proposta pelo projeto Jigsaw.

1.3 Terminologia linguística

Esta dissertação está escrita de acordo com o novo Acordo Ortográfico. Nesta dissertação o autor tenta, sempre que possível, utilizar terminologia de origem portuguesa. No entanto, tal nem sempre é possível, uma vezes porque é difícil encontrar o termo ou expressão correspondente e noutras, embora existindo tradução possível, essa não reflete o verdadeiro sentido do termo original na língua inglesa, geralmente com conotações fortemente orientadas às áreas das ciências informáticas ou a especificações técnicas escritas na língua inglesa.

A utilização de terminologia de origem inglesa acontece ao longo da dissertação pelas razões evocadas. De forma a assinalar essas situações, a terminologia inglesa utilizada aparece sempre destacada a itálico na primeira

ocorrência, apresentando sempre que seja justificável, uma explicação detalhada ou, como nota de rodapé, uma tradução possível na língua portuguesa.

Importa ainda destacar, no que diz respeito a referências bibliográficas, a existência de várias referências a bibliografia presente apenas em páginas na internet. A natureza do projeto, ainda numa fase inicial de definição e desenvolvimento, e o próprio modelo de desenvolvimento, fazem com que a informação exista essencialmente online, disponibilizada nas páginas do projeto ou por terceiros, ligados ao mesmo.

1.4 Conceitos base

É importante apresentar, de uma forma simples, um conjunto de conceitos e metodologias relacionadas com a linguagem Java para mais facilmente perceber o âmbito desta dissertação.

A linguagem Java foi apresentada, em 1995 [Horstmann 04], pela uma empresa Sun Microsystems e, a primeira versão oficial, 1.0, foi apresentada em 1996.

Desde o início, a linguagem Java foi mais do que uma mera linguagem de programação [Horstmann 04], tendo desde a versão inicial estado associada a um ambiente completo, com *API* e ambiente de execução próprios, formando aquilo que se designa por plataforma Java.

Em 1998 [JTML][JCP 10], o organismo intitulado *JCP – Java Community Process*, foi criado, cujo objetivo é i) gerir a definição da linguagem e plataforma Java; ii) fazer a gestão dos interesses múltiplos relativos à plataforma Java; iii) permitir uma evolução coerente do conjunto de especificações que definem a linguagem Java.

Atualmente, espera-se que a evolução da plataforma Java seja governada por este organismo.

Este organismo faz a gestão de um conjunto de propostas, provenientes dos associados, que pretendem ver incorporados na especificação Java metodologias

e ideias por eles defendidas. Podem ser associados, empresas, entidades sem fins lucrativos, nomeadamente instituições de ensino assim como entidades particulares [JCPPROC 09].

Estas propostas são apresentadas sobre a forma de *JSR*[2] *– Java Specification Request* – e são colocadas à votação segundo um processo definido nas normas do JCP e em resultados disso podem ou não ser aprovadas.

Uma vez aprovadas, passam a fazer parte do conjunto de especificações formais que definem todo o ambiente Java.

Para cada nova versão Java, são selecionadas as JSR aprovadas a incluir na nova versão. Este processo de definição de uma nova versão da plataforma Java pode ele próprio ser definido por uma JSR. Nem todas as especificações aprovadas fazem parte da plataforma Java embora sejam formalmente conhecidas como especificações Java.

As JSR são tipicamente definidas como especificações formais, sem implementações associadas. Para demonstração das especificações, é frequente ser apresentada uma implementação de referência[3]. Esta abordagem permite que as organizações possam adotar implementações variadas da mesma especificação, adaptadas a realidades distintas, mantendo mesmo assim a coerência com a especificação, isto é, respeitado a JSR.

É o conjunto de JSR's aprovadas e selecionadas para incluir as várias versões Java que definem toda a plataforma Java. Fora deste processo encontram-se as especificações iniciais, que existem desde o lançamento oficial da plataforma Java e que se tornaram parte integrante da especificação desde a sua origem.

Todo o processo segue uma metodologia que faz parte da própria definição da JCP.

Para garantir o melhor enquadramento e aplicabilidade da plataforma nos mais diversos ambientes, a plataforma Java foi dividida em 3 grandes áreas

2 JSR – Java Specification Request – poderá traduzir-se na língua portuguesa para Pedido de Especificação Java.

3 Tipicamente designada em inglês, no processo JCP por *RI – Reference Implementation*.

[Horstmann 04]: i) J2SE – plataforma base, destinada à implementação de sistemas generalistas; ii) JEE – destinada aos sistemas mais vocacionados para o mercado empresarial; iii) J2ME – destinada a ser utilizada em equipamentos tipicamente com poucos recursos computacionais, tais como telemóveis e caixas digitais de receção de televisão e de gravação de vídeo. As plataformas JEE e J2ME são, de forma oposta, derivadas da plataforma base J2SE. A plataforma JEE contêm toda a plataforma J2SE mais um conjunto de funcionalidades que têm aplicabilidade em ambientes empresariais. A plataforma J2ME contêm um sub-conjunto da plataforma J2SE mais um conjunto de funcionalidades de aplicabilidade em ambientes de baixos recursos computacionais.

Desta forma, quando nesta dissertação é referido o termo Java ou plataforma Java, pretende-se, salvo indicação em contrário, fazer referência à última versão, 6.0, do conjunto de especificações que definem a linguagem e plataforma Java base - J2SE 6.

1.5 Estrutura da tese

A dissertação está dividida em 7 capítulos.

No Capítulo 1 é realizada a apresentação da dissertação.

No Capítulo 2 é apresentado o que pode ser considerado como o estado da arte de modularidade em Java, apresentando várias abordagens atualmente assumidas como boas práticas para aplicabilidade de modularidade na plataforma Java.

No Capítulo 3 é apresentado o modelo de modularidade proposto no projeto Jigsaw.

No Capítulo 4 é apresentado o estudo sobre as implicações no desenvolvimento Java segundo a metodologia modular Jigsaw, estudando em pormenor a sua aplicabilidade em servidores aplicacionais que, pela sua complexidade tornam o estudo mais amplo e abrangente.

No Capítulo 5 são apresentadas um conjunto de alterações que visam

melhorar a proposta de modularidade Jigsaw, com base no resultado do estudo que vai ser realizado.

No Capítulo 6 é apresentada uma aplicação informática (prova de conceito) que ajuda a definir os módulos a partir de código fonte, facilitando a gestão de toda a nova metodologia aplicada a desenvolvimento de aplicações Java.

No Capítulo 7 são apresentadas as conclusões da dissertação, incluindo trabalhos futuros que possam resultar deste estudo.

Capítulo 2

Modularidade em Java – revisão do estado da arte

Este capítulo apresenta um conjunto de soluções que possibilitam a implementação de estratégias de desenvolvimento modular em sistemas de software desenvolvidos na plataforma Java.

2.1 Metodologias de modularidade

Existem várias metodologias para desenvolvimento de sistemas de software [Tate 04]. A modularidade é uma delas.

Não existe uma definição exata para modularidade em desenvolvimento de sistemas de software. Mesmo em linguagens de programação que não suportam objetivamente conceitos de modularidade, é geralmente possível aplicar práticas de modularidade, geralmente através de ferramentas auxiliares. As metodologias modulares estão muitas vezes implícitas em projetos de software, muitas das vezes sem que os arquitetos desses projetos tenham intencionalmente a convicção de aplicar essas metodologias – simplesmente resulta da necessidade natural de otimizar o processo, o que que por sua vez acaba por levar os arquitetos desses projetos a adotar direta ou indiretamente conceitos de modularidade.

Podemos no entanto identificar pontos importantes que caracterizam uma estratégia de programação modular [Kiczales 05][Bracha 92]:

- Existência de contextos funcionais, permitindo assim definir âmbitos funcionais orientados para a resolução de problemas.

- Interfaces perfeitamente definidos sendo esses os pontos de comunicação dentro do sistema.

- Interfaces como abstrações das implementações reais dos módulos, incluindo dados e funções, permitindo, uma vez definidas as interfaces dos módulos, efetuar alterações aos módulos, corrigindo erros ou re-implementando partes dos módulos para obtenção de vantagens, nomeadamente melhoria de performance computacional ou correção de erros, sem que os programadores tenham que alterar o resto do sistema.

- Capacidade integrada de gestão do ambiente por forma a satisfazer todas as necessidades em termos de interdependências entre módulos.

- Processo de juntar, de forma automática, todos os módulos por forma a obter um sistema funcional.

- Possibilidade de utilizar o mesmo módulo em vários locais distintos, incluindo noutros sistemas. A modularização de um sistema software deve ter como uma das vantagens, evitar a implementação das mesmas funcionalidades em sub-sistemas ou módulos diferentes.

Modularizar um projeto de software assume especial importância quando os projetos são desenvolvidos por mais do que uma equipa, situação muito comum em projetos desenvolvidos no âmbito do mercado empresarial. Desta forma, as empresas e organizações conseguem paralelizar processos, melhorar capacidade de resposta às solicitações do mercado e dos clientes e assim obter vantagens de processos.

Mesmo num modelo de desenvolvimento de apenas uma equipa ou em projetos onde a componente comercial não pesa, a modularidade apresenta vantagens reais relacionadas com isolamento funcional, organização de processos e identificação facilitada de pontos de falha [Turner 80].

Importa distinguir dois níveis de modularidade. Modularidade de

linguagem de programação e modularidade de sistemas. O primeiro define, na própria linguagem, conceitos modulares que são aplicados no processo de conceção de software. Por si só não representa um sistema modular, apenas uma metodologia de programação modular. O segundo, apresenta todo um conjunto de facilidades que permitem que todo um sistema de software possa ser modularizado no ato da sua conceção e no ato da sua execução. Não implica necessariamente que a ou as linguagens de programação utilizadas suportem conceitos de modularidade. Este pode ainda dividir-se em dois sub-níveis: i) modularidade da plataforma: a própria plataforma base suporte modularidade e está modularizada; ii) modularidade de aplicação: quando existem mecanismos para implementar sistemas de software modulares, independentemente da plataforma base ser ou não modular.

Idealmente, uma metodologia modular deverá abranger todos os níveis.

2.2 Modularidade em Java

A Java é uma linguagem orientada por objetos e não suporta na definição da linguagem e da máquina virtual Java um sistema de módulos [Strniša 07]. Apresenta no entanto alguns conceitos que facilitam a implementação de alguns aspetos que caracterizam sistemas modulares, próprios do facto de ser uma linguagem estruturada.

Para além desses conceitos base, ao longo dos anos, foram efetuados estudos e implementados sistemas para facilitar o processo de modularidade na plataforma Java. Atualmente, existem 3 abordagens importantes na forma de JSR's: i) JSR 277 - Java™ Module System; ii) JSR 291 - Dynamic Component Support for Java™ SE; iii) JSR 294 - Improved Modularity Support in the Java™ Programming Language.

Nas próximas subsecções são abordadas com mais detalhe as estratégias de modularidade atrás enunciadas.

2.2.1 Sistema de *packages*

Os *packages*[4] [Gosling 05] são utilizados frequentemente para organizar sistemas desenvolvidos em Java, proporcionando algumas funcionalidades modulares como sejam isolamento funcional e abstração de dados [Strniša 07].

Cada package tem o próprio conjunto de classes, que ficam completamente definidas através do nome do package. Desta forma, é possível existirem classes com o mesmo nome em packages diferentes.

Os packages podem ser armazenados em sistemas de ficheiros, bases de dados ou em outros tipos de armazenagem [Gosling 05].

Os packages representam, depois das classes, a unidade mais baixa de organização de código em Java, não representando no entanto um verdadeiro processo de implementação de modularidade.

2.2.2 Ficheiros *JAR, WAR e EAR*

Na plataforma Java, a estratégia utilizada pelo ambiente de execução para carregamento das classes é carregar as classes para o ambiente de execução à medida que vão sendo necessárias. Esse processo é realizado por um gestor de carregamento de classes [Gosling 05] designado por *ClassLoader*[5], mapeando as classes em memória. Podem existir mais do que um ClassLoader no ambiente Java, providenciando formas distintas de carregar as classes no ambiente Java a partir de fontes distintas. O ambiente de execução ambiente Java pode ter mais do que um ClassLoader ativo. O ClassLoader base, utilizado sempre que não se utilizam ClassLoaders especializados, carrega e mapeia em memória as classes através do nome completo das classes, que é composto pelo nome do package e o nome da classe.

---

4 Packages é o termo utilizado na especificação Java. Em português tem como tradução pacotes. Por uma questão de alinhamento com a especificação o termo package será utilizado durante a dissertação.

5 ClassLoader não tem uma tradução direta para língua portuguesa. Uma tradução possível poderá ser carregador de classes. Trata-se de um termo que faz parte da especificação Java e que será utilizado durante a dissertação por questões de simplicidade e correspondência com a especificação.

As classes Java podem ser carregadas do sistema de ficheiros a partir de ficheiros individualizados para cada classe, distribuídos em diretorias que representam os packages ou a partir de outros tipos de fontes, nomeadamente ficheiros *JAR, WAR e EAR*. Os ficheiros *JAR,* em inglês, ***J**ava **Ar**chive [JARSPEC],* representam arquivos Java na especificação Java e são atualmente a base de estruturação e distribuição mais utilizada de aplicações Java. Os ficheiros *WAR,* em inglês, ***W**eb **A**pplication a**R**chive* e os ficheiros *EAR,* em inglês, ***E**nterprise **Ar**chive,* são ficheiros JAR, definidos na especificação JEE [Crawford 05], com particularidades que os tornam importantes no ambiente JEE, nomeadamente no que diz respeito à existência de informação do tipo meta-informação adicional que permite utilizar estes ficheiros de forma eficiente em ambientes servidores [Crawford 05]. Desta forma, a explicação destes ficheiros (JAR, WAR e EAR) estará centrada nos ficheiros JAR.

Entende-se por meta-informação, neste contexto, um conjunto de informação que é utilizada para descrever de forma sistematizada e seguindo normas, o conteúdo dos ficheiros onde está contida, neste caso os ficheiros JAR, WAR e EAR.

Os ficheiros JAR são contentores [JARSPEC], que podem conter tipos diferentes de dados, nomeadamente classes, imagens, documentos e outros tipos de dados. São ficheiros no formato ZIP e geralmente têm a extensão ".jar" [Darwin 04]. Um ficheiro num formato ZIP é um ficheiro que apresenta características especiais, nomeadamente poder conter outros ficheiros e suportar compressão.

Estes ficheiros, representam, depois dos packages, a unidade mais baixa de organização e modularização do Java, permitindo:

- Distribuição de vários packages num único módulo.
- Forma otimizada de distribuição de módulos – Os ficheiros são uma variação dos ficheiros com formato ZIP e desta forma, o seu tamanho pode ser reduzido por processos de compressão.

Estes ficheiros e os processos associados apresentam no entanto limitações: [Gosling 05]:

- Não permitem ter no sistema vários packages com o mesmo nome, mesmo que definidos em ficheiros JAR distintos.

- Não suportam gestão de versões.

- A informação de apoio – meta-informação – é muito simplificada e não fornece informação que possa ter aplicabilidade num sistema modular. Os ficheiros WAR e EAR apresentam um conjunto mais alargado de normas de definição de meta-informação do que os ficheiros JAR sendo que, no entanto, essa meta-informação tem aplicabilidade apenas na plataforma JEE [Crawford 05].

- Não se relacionam uns com os outros.

- Potenciam a ocorrência de problemas e erros no desenvolvimento e distribuição de sistemas de software em Java, conhecidos pela expressão "*JAR Hell*", que poderá traduzir-se para português por "Inferno dos ficheiros JAR" e que pode descrever-se [Strniša 07] por, na plataforma Java, não existir atualmente nada que impeça a existência de ficheiros JAR a disponibilizar versões diferentes dos mesmos packages, sem que existam alertas para tal situação, levando a situações imprevisíveis na execução das aplicações Java. Quando o ClassLoader base da plataforma carrega para memória uma classe, esta fica mapeada pelo nome completo. Se existir uma versão diferente da mesma classe, noutro ficheiro JAR, este não será carregado nem utilizada uma vez que o ClassLoader assume que já está presente em memória. No entanto, esta poderá ser a classe que o ClassLoader devia carregar para satisfazer as dependências no ambiente de execução. Esta forma de funcionamento invalida a possibilidade de utilizar estes ficheiros para implementação de modularidade de forma adequada em Java. O problema surge quando, diferentes componentes necessitam de diferentes versões de um outro componente. Embora este terceiro componente possa estar representado em duas versões no sistema, possivelmente através de dois ficheiros JAR, apenas uma das versões é utilizada, levando a resultados imprevisíveis e a figura seguinte modela a forma como a máquina virtual pode criar situações de inconsistência.

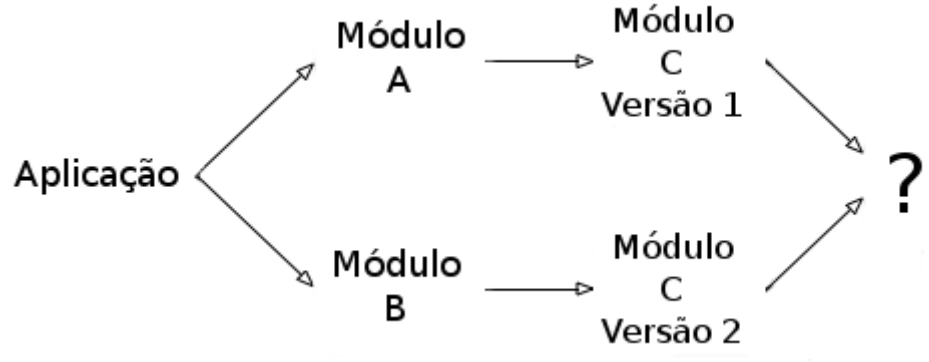


Figura 2.2.1: O problema JAR HELL

O problema pode ser descrito da seguinte forma. Assumindo que uma aplicação é composta por uma conjunto de classes. Estas classes dependem por sua vez de funcionalidades presentes em 2 ficheiros JAR distintos, ou módulos, A e B. Por sua vez, estes ficheiros dependem de funcionalidades presentes num outro ficheiro JAR – C – sendo que A depende da versão 1 do ficheiro C e B depende da versão 2 do ficheiro C. Durante a execução e devido ao facto de a plataforma Java e os gestores de classes não suportarem múltiplas versões das mesmas classes, independentemente de ambas as versões do ficheiro C estarem presentes no sistema, apenas uma será carregada para memória, sendo que ou o ficheiro A ou o ficheiro B verão as suas dependências satisfeitas incorretamente, causando resultados imprevisíveis, potenciando perdas de dados e outras situações criticas durante a execução da aplicação. Esta situação não é detetável no ato de compilação dos programas.

A utilização destes ficheiros facilita a implementação e distribuição de sistemas na plataforma Java. No entanto, esta estratégia de implementação e distribuição de sistemas software desenvolvidos em Java não representa um verdadeiro sistema modular e representa riscos importantes que devem ser eliminados.

São utilizados para distribuição de aplicações, de forma modular e para implementar sistemas de módulos ou *plugins*[6]. São exemplos os sistemas

6 Não existe uma tradução simples para este termo de origem inglesa. No entanto, no contexto de desenvolvimento de sistemas software, nomeadamente de sistemas software desenvolvidos na plataforma Java, o termo plugin tem conotação com o termo módulo que adiciona funcionalidade ao sistema.

servidores de aplicações e sistemas de aplicações complexos.

A plataforma JEE, utiliza-os (ficheiros WAR e EAR) como forma de distribuição[7] de módulos na infra-estrutura dos servidores de aplicações JEE. Estas funcionalidades são conseguidas recorrendo a lógica adicional disponibilizada por funções dedicadas na plataforma JEE. No entanto e mais uma vez, é possível ocorrerem os mesmos erros típicos do JAR HELL.

Estes ficheiros não representam assim um modelo para implementação de sistemas modulares.

2.2.3 JSR 277

A JSR 277 foi apresentada em 2006, como resultado de um grupo de trabalho criado em 2005, e define um conjunto de especificações que apresentam um modelo de modularidade para a plataforma Java [JSR277].

A especificação define [JSR277]:

- Um formato de distribuição – Módulo Java – para distribuição de packages Java. Este formato tem meta-informação sobre o conteúdo do módulo e as dependências com outros módulos. Este formato tem como base os ficheiros JAR, tendo no entanto um conjunto de meta-informação que permite fornecer toda a informação necessária sobre o conteúdo do módulo (ficheiro JAR).

- Um processo de gestão de versões de módulos.

- Um processo de gestão de bibliotecas de módulos que permita guardar e procurar as versões corretas dos módulos.

- Suporte no ambiente, através de ClassLoaders especializados e outras funcionalidades, para as funcionalidades previstas nos pontos anteriores.

- Capacidade de coexistência e possibilidade de migração do sistema atualmente existente que se baseia em ficheiros JAR.

7 Na especificação Java (J2SE, JEE e J2ME) o termo em inglês adotado é geralmente *deploy.*

A JSR 277 apresenta um sistema completo de modularidade para a plataforma Java. É uma especificação técnica e muito relacionada com uma implementação real. Os JSR são tipicamente apresentados como especificações abstraídas de implementações reais e que permitem geralmente implementações distintas respeitando no entanto a especificação, o que não acontece com a JSR 277, que representa a própria implementação.

É importante indicar que o JSR não prevê qualquer alteração na linguagem Java pelo que, apesar de disponibilizar uma forma de modularizar software escrito em Java, a linguagem Java e toda a programação em Java continuarão a existir sem conceitos de modularidade.

Esta JSR teve uma aceitação baixa no JCP e não foi, até à data, aceite como especificação. Desta forma, a especificação JSR 277 não foi selecionada nem se prevê que possa vir a ser, para ser incorporada na próxima versão da plataforma Java, Java 7 [JSR277]. No entanto, todos os conceitos de modularidade estão presentes neste JSR daí a sua importância neste estudo, uma vez que, como será apresentado mais à frente na dissertação, os resultados deste JSR vão ser utilizados, para uma futura base de modularidade na plataforma Java.

2.2.4 JSR 291

A *Open Services Gateway Initiative* (OSGi) é uma organização composta por empresas importantes no mercado das tecnologias da informação [OSGI], nomeadamente, Sun Microsystems Inc., IBM Corporation, Siemens AG, RedHat, Oracle Corporation, entre outras, cujo objetivo é implementar e promover a utilização em ambientes Java de um verdadeiro sistema modular de distribuição e utilização de componentes de software – módulos.

A organização definiu um conjunto de especificações, intituladas de "OSGi Service Platform Release", sendo atualmente a versão 4 a mais recente da especificação.

Em 2006, a organização submeteu ao JCP uma JSR – JSR 291: Dynamic Component Support for Java™ SE – sendo que este JSR representa o pedido de inclusão da especificação OSGi na plataforma Java através do JCP.

Um dos argumentos para a proposta da JSR é permitir ter uma especificação sobre um sistema modular na versão da plataforma Java J2SE 6 enquanto se procede ao estudo da melhor solução para a J2SE 7, em desenvolvimento [JSR291].

O JSR, após votação, foi aceite como especificação. No entanto, até à data não foi selecionado para ser incluído na plataforma Java embora esteja a ser utilizado em projetos importantes, como é o caso da plataforma de desenvolvimento Eclipse [ECLIPSE].

O modelo de modularidade assenta o conceito numa arquitetura de serviços [Ahn 06], agrupados em módulos, que os exportam. Esses serviços são utilizados por outros módulos. Os serviços são distribuídos em packages que são designados de *bundles*[8] na especificação OSGi [OSGi 09].

Os bundles são entidades que atuam de forma isolada no ambiente OSGi e que são geridos por diferentes ClassLoaders, garantindo assim isolamento entre módulos [Geoffray 08].

Os bundles fazem uso de um registo central, também fazendo parte do modelo OSGi, que serve de diretório de serviços [OSGi 09].

Desta forma, os bundles exportam serviços próprios através de um registo central e, quando necessitam de utilizar serviços de outros bundles, recorrem-se do registo central para localizar e utilizar esses serviços de outros bundles.

A utilização desses serviços é realizada por evocação direta de métodos Java, garantindo performance.

A atual arquitetura OSGi consiste em 4 camadas, que executam no topo do ambiente de execução – a plataforma Java, tal como se representa na figura 2.2.1, que está baseada numa figura da arquitetura presente na documentação OSGi [OSGi 09].

8 Bundle é o termo indicado na especificação para designar um conjunto de serviços pertencentes a um mesmo módulo. Nesta utilização, bundle pode ser traduzido como "pacote" na língua portuguesa.

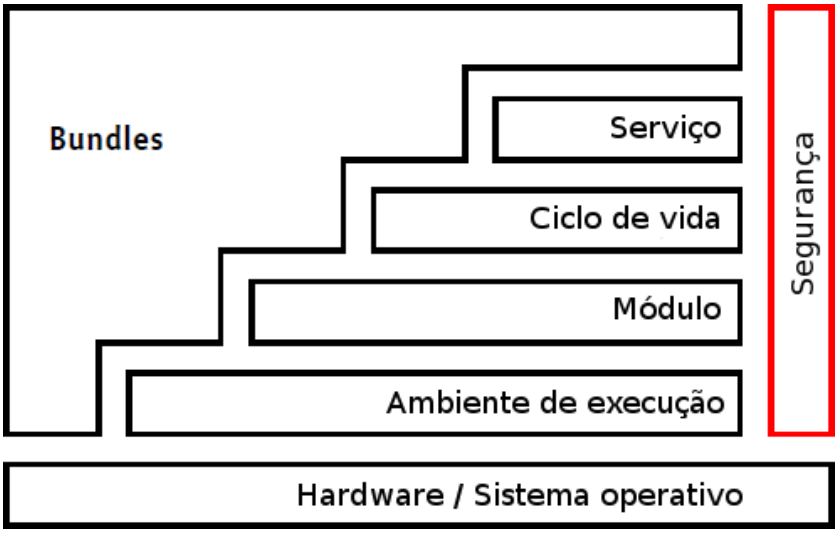


Figura 2.2.2: Modelo OSGi

A camada de segurança implementa o conjunto de funcionalidades necessárias para garantir segurança. Estas funcionalidades são implementadas recorrendo aos processos de segurança suportados na plataforma Java e estendendo as capacidades de segurança da plataforma Java nas áreas que esta não cobre. As melhorias podem descrever-se por: i) definição de um sistema de ficheiros JAR seguro, recorrendo a processos digitais de assinatura e a meta-informação adicional; ii) APIs de interação com o sistema de segurança da plataforma Java para facilitar o processo de gestão de políticas de segurança durante a execução das aplicações.

A camada de módulos implementa o conjunto de funcionalidades necessárias para suportar modularidade em Java, nomeadamente: i) especificação de como os módulos (bundles) definem os serviços que exportam e que importam; ii) gestão de versões; iii) modelo de criação dos módulos; iv) modelo de distribuição de módulos.

A camada de ciclo de vida providência uma API consistente que permite utilizar de forma unificada as funcionalidades disponibilizadas pelas camadas de segurança e de módulos.

Por último, a camada de serviço representa o agregador de serviços e está diretamente relacionada com a camada de ciclo de vida. Esta camada (diretório)

é utilizada para proceder ao registo de serviços, localização de serviços, ativação do serviços, libertação de serviços, entre outras funcionalidades.

As 4 camadas são executadas na plataforma Java. O conceito agregador é o bundle.

O modelo de modularidade definido pela OSGi apresenta vantagens técnicas importantes relativamente a outros modelos de modularidade conhecidos.

O modelo não necessita de alterações à linguagem Java ou à máquina virtual Java o que se traduz numa vantagem ao permitir aplicabilidade imediata do modelo.

As aplicações OSGi são implementadas e executadas na plataforma Java como qualquer outra aplicação Java. Este conceito é importante dado que o ambiente OSGi disponibiliza um ambiente modular às aplicações desenvolvidas segundo a especificação e que são executadas dentro do ambiente OSGi mas o próprio ambiente de execução Java (máquina virtual e linguagem) não é modular. Da mesma forma, as aplicações, na sua escrita, não têm relação direta com modularidade. Para utilizar este ambiente, é necessário utilizar o conjunto de bibliotecas OSGi que disponibilizam toda a funcionalidade de gestão de módulos.

A metodologia OSGi, tendo como patrocinadores empresas importantes, foi adotada por alguns projetos importantes sendo que mesmo assim a sua adoção tem sido lenta, nomeadamente porque necessita de todo um conjunto de software adicional para ser implementada e o processo de modularidade não é algo assumido na plataforma Java, o que impede que a maioria dos projetos a adote.

2.2.5 JSR 294

O JSR 294 foi apresentado em 2007, por um grupo de trabalho constituído em 2006 e tem uma abordagem diferente das propostas pelos JSR 277 e JSR 291 [JSR294].

O objetivo principal da especificação é, através de alterações mínimas na especificação da linguagem Java e na especificação da máquina virtual Java, dotar a plataforma Java de capacidades de modularidade.

Este processo é conseguido aplicando as mesmas metodologias de especificação de linguagem e de máquina virtual já existentes, como por exemplo as de controlo de acesso ás classes – *private, public, protected* – para expor apenas as classes e interfaces que se pretende.

A JSR 294 define uma nova entidade na especificação Java, chamada *superpackage*[9]. Um superpackage contêm uma lista de classes e interfaces e, destas classes, as públicas são marcadas como exportáveis, ficando assim disponíveis a ser utilizadas fora da superpackage [JSR294].

A especificação tem como objetivos principais [JSR294]:

- utilizar e estender o modelo de controlo de acesso definido pela especificação de linguagem Java e especificação de máquina virtual Java para suportar domínios mais abrangentes.

- permitir que os mecanismos de controlo de acesso possam ser simples e consistentes com os mecanismos atualmente existentes.

- permitir que políticas de gestão de acesso sejam definidas nos ficheiros de código fonte por forma a, logo no ato da compilação, poderem ser aplicadas políticas de controlo.

- permitir que o novo modelo possa ser utilizado para a criação de módulos mas também, ser possível utiliza-lo de forma independente, não necessitando da existência de nenhum modelo adicional de criação de módulos nem de alterações aos modelos já existentes.

A especificação não apresenta assim um sistema modular, com suporte para a criação de módulos, de gestão de versões, de gestão de bibliotecas ou de outros elementos necessários à implementação de um verdadeiro sistema modular.

9 Este termo surge na especificação do JSR 294 e em português pode traduzir-se como super pacote embora do ponto de vista de utilização deva ser utilizado o termo em inglês.

Delega essas funcionalidades para outras especificações que podem ser por exemplo as especificações JSR 277 e JSR 291.

Apresenta apenas o conjunto de alterações mínimas a efetuar à especificação da linguagem Java e à especificação da máquina virtual Java para proporcionar os elementos nucleares para implementação de mecanismos de modularidade [JSR294] na própria linguagem Java. É, dos modelos apresentados, o único que prevê a introdução de conceitos de modularidade na própria linguagem Java.

O JSR 294, para ser incorporado na plataforma Java, necessita de alterações à linguagem Java e à máquina virtual Java.

Não existe atualmente nenhuma implementação de referência do JSR 294.

O JSR 294 foi aceite[10] para ser incorporada na próxima versão Java – Java 7.

2.2.6 Plugins

Apresenta-se a metodologia de plugins como uma metodologia de implementação de modularidade apenas para efeitos comparativos. Na verdade, a abordagem de plugins utiliza sempre, de uma forma ou de outra, ideias equivalentes às metodologias apresentadas anteriormente.

São utilizados geralmente ficheiros JAR ou variantes para distribuição e gestão dos módulos e de alguma forma, é realizada uma gestão de versões, sempre de forma proprietária[11], recorrendo a implementações de gestão de plugins próprias. O projeto Netbeans [NETBEANS] utiliza por exemplo um sistema de plugins, com gestão autónoma desses plugins e de todas as dependências entre eles. No entanto, esta metodologia, como todas as outras metodologias baseadas em plugins mais comuns, só faz sentido no ambiente de desenvolvimento Netbeans, com todas as bibliotecas de código necessárias à gestão dos plugins.

10 Embora tenha sido aceite, uma falha de processo no JCP faz com que este JSR esteja marcado como "Inativo".

11 Independentemente de poderem existir especificações dos modelos de plugins, eles não são especificados pelo processo JCP pelo que se consideram abordagens proprietárias.

Os sistemas de plugins mais comuns possibilitam a implementação de sistemas modulares mas a aplicabilidade destas metodologias de forma transversal, incluindo para modularizar a própria plataforma Java, é algo que embora possível do ponto de vista de puro desenvolvimento de software, não seria simples dado que ia adicionar lógica de gestão de plugins complexa na própria plataforma Java e limitava a aplicabilidade de outras metodologias modulares.

2.2.7 Aspect-oriented programming

Nos anos mais recentes, surgiu uma nova tendência, enquadrada em metodologias de programação estruturada, designada por AOP – Aspect-Oriented Programming[12] – que em última instância permite modularizar sistemas de software. Em português pode designar-se por programação orientada a aspetos.

A metodologia foi criada pela equipa de desenvolvimento e investigação Xerox PARC, da empresa Xerox.

Embora não se pretenda apresentar em detalhe a metodologia, a sua apresentação é importante para compreender de que forma uma das mais recentes tendências de metodologias de programação comparam no âmbito do estudo defendido na dissertação.

A AOP representa uma nova metodologia de implementação de aplicações, que pode ser considerada como uma evolução relativamente às metodologias orientadas por objetos [Bernardi 09].

A AOP introduz conceitos novos, dos quais se destacam:

- Objetivos (*Concern*) – as aplicações podem decompor-se em objetivos, funcionais ou não funcionais, que têm que ser respondidos e concretizados pelas aplicações.

12 Por simplicidade de associação, durante a dissertação, a sigla em inglês – AOP – será utilizada para referenciar a programação orientada a aspetos.

- Objetivos transversais (*Cross-cutting concerns*) – Representam objetivos que estão identificados no âmbito da aplicação como sendo transversais à aplicação. A identificação destes objetivos permite isolar os outros objetivos destes, permitindo um apuramento do código fonte em função daquilo que necessita fazer por si só, e não do que necessita fazer, em função do resto do sistema.

- Aspetos (*Aspect*) – Representam a aplicabilidade de regras que aplicam os objetivos transversais à aplicação.

A AOP encara o código fonte das aplicações de forma ampla, disponibilizando os mecanismos e metodologias para identificar, modularizar e implementar os objetivos transversais no código fonte das aplicações através dos aspetos [Cazzola 05].

A metodologia não existe por si só, isto é, não permite o desenvolvimento direto de aplicações, necessitando sempre de uma linguagem de suporte.

Foi nesse sentido que a mesma equipa que criou o conceito criou também uma aplicação, a AspectJ, que permite aplicar a metodologia ao desenvolvimento em Java [Kiczales 01]. Esta aplicação é atualmente mantida no âmbito do projeto Eclipse [ASPECTJ].

Importa referir que a AOP é uma metodologia que não é exclusiva do Java e tem sido aplicada com grande sucesso noutras linguagens, nomeadamente linguagens C e C#.

A AspectJ permite a definição dos vários blocos funcionais de uma aplicação, dividindo uma aplicação em módulos. O código Java não necessita ter qualquer conhecimento da modulação, uma vez que esta é realizada externamente. AspectJ não especifica alterações à linguagem Java nem define qualquer associação concreta com módulos.

É no ato de execução da aplicação que o ambiente funcionalmente modularizado funciona, permitindo que os módulos não transversais (*concerns*) utilizem de forma transparente módulos transversais (*Cross-cutting concerns).*

A AOP representa uma evolução relativamente às linguagens de programação mais utilizadas uma vez que aproxima ainda mais o desenvolvimento aplicacional dos problemas reais que se pretendem ver resolvidos, através de uma estruturação natural que se pode designar por modular.

Têm sido realizados estudos com a utilização das metodologias UML aplicadas à AOP, como em [Cazzola 05] permitindo facilitar ainda mais todo o processo de aplicação da AOP.

O AOP e a implementação para o Java, o AspectJ, não definem características adicionais associadas a módulos que foram sendo identificadas neste capítulo, nomeadamente: i) versões; ii) dependências entre módulos; iii) repositórios de módulos; iv) segurança entre módulos na linguagem Java. Definem apenas todo um sistema que age de forma estruturada e modular, na sua globalidade.

O AOP não permite implementar de forma direta todo um sistema modular, tal como pretendemos ver implementado. Não permite garantir a consistência de todos os componentes de um sistema aplicacional, através de processos automáticos de resolução de dependências e recorrendo por exemplo a versões. Nem tem como objetivo esses requisitos.

2.3 Comparação das estratégias de modularidade apresentadas

Das estratégias analisadas, podemos comparar aspetos que permitem melhor enquadrar cada uma.

A linguagem Java, permite, na sua definição, com utilização de packages, implementar separação funcional e abstração de dados. Os packages fazem parte da especificação da linguagem Java.

Com a utilização de ficheiros JAR, WAR e EAR, é possível agrupar packages por áreas funcionais, muitas das vezes associados à designação de plugins, assim como definir, de forma consistente mas limitada, o conteúdo dos ficheiros

através de meta-informação. Estes ficheiros apresentam ainda facilidades de distribuição de aplicações e de criação de sistemas modulares, como por exemplo servidores aplicacionais.

Tanto os ficheiros JAR, WAR e EAR assim como os packages fazem parte da especificação Java.

Estas facilidades não são suficientes para a implementação de modularidade na plataforma Java. Existem assim 3 especificações (JSR 277, JSR 291 e JSR 294) que procuram de facto apresentar abordagens distintas para dotar a plataforma Java de capacidade de modularidade. Existe também uma nova metodologia que embora não tenha como objetivo direto dotar o Java com modularidade, permite implementar modularidade em Java.

A tabela seguinte apresenta o conjunto de facilidades que cada uma das definições especifica:

	JSR 277	JSR 291	JSR 294	AOP
Abstração de implementação e de dados	S	S	S	S
Conceito de módulo	S	S	S	S[13]
Conceito modular na especificação Java			S	
Conceito modular na máquina virtual Java			S	
Políticas de segurança aplicáveis aos módulos na especificação Java			S	
Repositório de módulos	S	S		
Características de gestão de versões	S	S		
Define todos os sistemas necessários à implementação de um sistema modular	S	S		
Existe implementação de referências	S	S		S
JSR aprovado pelo JCP		S	S	

Tabela 2.3.1: Tabela comparativa da JSR 277, JSR 291, JSR 294 e AOP

13 Na AOP, o conceito de módulo não é algo que se possa identificar programaticamente mas o conceito modular e de módulo está presente.

A letra “S” significa, na tabela a existência da propriedade referenciada.

As especificações apresentadas definem, através da JSR 277, um sistema completo de suporte a modularidade que pode ser utilizado em sistemas desenvolvidos sobre a plataforma Java. Não necessitam de alterações à linguagem ou máquina virtual Java. Através desta JSR, a plataforma Java não está modularizada. Trata-se de uma JSR que não foi aprovada no processo JCP.

Através da JSR 291, definem um sistema completo de suporte a modularidade que pode ser utilizado em sistemas desenvolvidos sobre a plataforma Java. Não necessitam de alterações à linguagem ou máquina virtual Java. Através desta JSR, a plataforma Java não está modularizada. Trata-se de uma JSR que foi aceite no processo JCP podendo ser uma candidata a fazer parte de uma próxima versão Java. Tem forte apoio por parte de algumas empresas importantes nas áreas das tecnologias de informação.

Através da JSR 294, definem uma especificação que define um conjunto de alterações à definição da linguagem Java e da máquina virtual Java e cujo objetivo é definir as bases para a implementação de metodologias modulares na própria linguagem Java e por sua vez, em toda a plataforma. Não define o sistema modular mas, os sistemas modulares que adotem esta especificação conseguirão implementar modularidade nos sistemas desenvolvidos sobre a plataforma Java e também na própria plataforma base, permitindo assim modularizar a própria plataforma. Não existe atualmente qualquer implementação de referência desta especificação.

Por último, foi abordada uma nova metodologia, a AOP, que se enquadra nas metodologias de programação modulares. Esta metodologia tem sido estudada em profundidade e apresenta uma nova abordagem de desenvolvimento modular, quase de forma transparente. É simples, não implica alteração no código Java, permite simplificação do código Java mas o foco está orientado aos aspetos funcionais e não funcionais de uma aplicação como um todo, numa aplicação e, não define aspetos mais básicos e técnicos como sejam repositórios de módulos, versões de módulos, resolução de interdependências de módulos, entre outros.

2.4 Problemas da modularidade em Java

As estratégias apresentadas procuram, de uma forma mais completa nuns casos e noutros, de forma mais simples, resolver o problemas relacionados com a ausência de capacidades de modularização no Java.

Podemos agora identificar esses problemas de forma clara:

- A especificação da linguagem Java e a especificação da máquina virtual Java não suportam o conceito de modularidade.
- A plataforma Java não está modularizada.
- Para a criação de sistemas software modulares na plataforma Java, é necessária a utilização de ambientes especializados, desenvolvidos por entidades com interesses divergentes, com metodologias próprias, existindo várias estratégias, sendo elas incompatíveis entre si.
- A plataforma JEE suporta o conceito de módulos, sendo no entanto e mais uma vez uma estratégia especifica do JEE, com fortes limitações em vários aspetos importantes, nomeadamente gestão de versões e repositório central de módulos.
- A complexidade de sistemas software desenvolvidos, nomeadamente para implementar plugins, tende a provocar uma maior esforço na distribuição e execução das aplicações, nomeadamente sistemas servidores aplicacionais, não evitando a ocorrência de falhas e erros graves, nomeadamente o de colisão de diferentes versões de packages num mesmo ambiente de execução, tal como acontece no conhecido processo errático "JAR Hell" que, dado a forma como o ambiente carrega as classes para execução, levam a problemas graves de utilização de versões de packages diferentes do esperado, situações que só se verificam geralmente no ato de execução e cuja deteção é difícil.

E por último, não propriamente um problema, mas sim uma necessidade, a ideia abrangente no JCP – e respetivos associados – da necessidade efetiva de dotar o Java de capacidades modulares completas, abrangentes e que sejam

aceites de forma suficientemente consensual no Java.

2.5 Conclusões

Torna-se claro que a plataforma Java não tem suporte na linguagem e máquina virtual que permita implementar sistemas modulares de forma consistente.

Verifica-se um esforço importante de várias entidades, ligadas a mundo empresarial e ao mundo académico, no sentido de dotar a plataforma Java de capacidade de modularização.

Na tabela seguinte, são apresentados os aspetos positivos e menos positivos de cada uma das especificações apresentadas.

	Aspetos positivos	Aspetos negativos
JSR 277	Especificação de um sistema que implementa os conceitos necessários à implementação de sistemas modulares; Não necessita de alterações na especificação da linguagem Java nem na máquina virtual Java;	Não está definida na especificação Java nem na máquina virtual Java; Não permite modularidade na máquina virtual nem na plataforma base Java; JSR não aprovada pelo JCP;
JSR 291	Especificação abrangente, que define todo o ambiente necessário à implementação de sistemas modulares; Especificação suportada por grupos empresariais importantes; Organismo dedicado à gestão da definição e implementação; Não necessita de alterações na especificação da linguagem Java nem na máquina virtual Java;	Não está definida na especificação Java nem na máquina virtual Java; Não permite modularidade na máquina virtual nem na plataforma base Java;
JSR 294	Define o conceito de modularidade nas especificações da linguagem Java e da máquina	Exige alterações às especificações respetivamente da

	virtual Java; Possibilita a implementação de modularidade na própria plataforma Java;	linguagem Java e da máquina virtual Java; Não define o ambiente nem o conjunto de sistemas necessários à implementação de um sistema modular[14];
AOP	Modulariza as aplicações identificando os requisitos e objetivos destas e definindo módulos que interagem entre si de forma transparente e deixam que o desenvolvimento se foque nos problemas que tem que resolver.	É uma metodologia nova, que embora não altere o a especificação da linguagem Java, é aplicada utilizando uma definição externa e processos de suporte externos; Modularidade está implícita mas o foco não é na verdade a definição de módulos, deixando desta forma de fora um conjunto de propriedades que se pretendem ver como características num sistema modular: repositórios de módulos, gestão automática de interdependências entre módulos e versões de módulos.

Tabela 2.5.1: Tabela comparativa das vantagens e desvantagens da JSR 277, JSR 291, JSR 294 e AOP

Da consulta da tabela destaca-se o facto de nenhuma das especificações apresentar simultaneamente: i) um modelo de modularidade para a máquina virtual Java e especificação Java; ii) um modelo para sistemas aplicacionais desenvolvidos na plataforma Java; iii) um modelo para os sistemas e APIs necessários para implementar modularidade.

Espera-se agora que, da mesma forma que outros conceitos foram incorporados na plataforma Java, um modelo robusto de modularidade e aceite de forma unânime possa ser também incorporado, capaz de proporcionar

14 Este aspeto pode, por outro lado, não ser considerado um aspeto negativo uma vez que o facto de não definir o conjunto de sistemas necessários à implementação, permite espaço para implementações variadas, respeitando no entanto a especificação.

modularidade não só nos sistemas desenvolvidos na plataforma Java mas também modularidade na própria plataforma Java, na linguagem Java e na máquina virtual Java.

Capítulo 3

Projeto Jigsaw

Este capítulo apresenta o projeto Jigsaw, de forma detalhada.

No fim deste capítulo, deve ser possível ter uma visão:

- Dos objetivos e âmbito do projeto.
- Da metodologia modular proposta.
- Dos processos base implementados pelo projeto.
- Das alterações necessárias ao JDK[15], máquina virtual, linguagem e outros elementos, para implementar o sistema de módulos proposto.

3.1 Introdução

O projeto Jigsaw é um projeto de desenvolvimento de software, promovido pela SUN, cujas funcionalidades pretende incluir na próxima versão do Java, JDK 7.

O projeto está a ser desenvolvido, em paralelo, como um sub-projeto do projeto principal de desenvolvimento JDK 7. As funcionalidades em desenvolvimento ainda não foram incorporadas na versão principal de

15 JDK – em inglês, Java Development Kit, que pode traduzir-se por ambiente de desenvolvimento Java, representa em termos de linguagem Java, a plataforma oficial Java, dedicada ao desenvolvimento de programas Java.

desenvolvimento do JDK 7. Exceto se existirem fatores maiores que impossibilitem a inclusão do sub-projeto Jigsaw no projeto principal, prevê-se a sua inclusão quando o desenvolvimento do JDK 7 estiver já numa fase avançada para que, quando o JDK 7 entrar em produção, possa incorporar já as facilidades propostas pelo Jigsaw.

A sua criação resulta da necessidade identificada pela SUN de adotar metodologias modulares na sua máquina virtual JDK [JIGSAW] e na plataforma Java, para daí obter vantagens técnicas e operacionais e do facto de, tal como se demonstra no Capítulo 2, não existir atualmente um modelo consensual de metodologia modular para ambientes Java que possa ser utilizado para desenvolver a própria plataforma base Java.

O estudo do projeto Jigsaw baseia-se em grande parte, na informação recolhida a partir de fontes internas ao projeto, diretamente ligadas ao desenvolvimento do mesmo. Atualmente, existe apenas uma versão em formato executável capaz de ser testada, que funciona apenas no sistema operativo Linux Ubuntu[16] [JIGSAW]. Trata-se de uma versão sem qualquer objetivo de ser estável, sendo apenas uma versão muito instável mas que permite demonstrar o conceito em funcionamento. É com base nesta versão que todos os exemplos e que demonstrações serão realizadas no decurso desta dissertação.

Durante a escrita desta dissertação, todas as alterações apresentadas, que são necessárias no âmbito da máquina virtual Java e da especificação de linguagem Java, ainda não estão completamente fechadas no projeto Jigsaw pelo que até à versão final, podem sofrer correções. Esta dissertação apresenta o trabalho realizado pelo projeto até à data, assim como o resultado de informação obtida das equipas que estão a desenvolver o projeto.

3.2 Objetivos

A SUN, com o projeto Jigsaw, tem como principal objetivo dotar a plataforma Java, versão SE, de capacidades modulares e de modularizar a própria plataforma.

16 Linux Ubuntu é uma distribuição Linux que pode ser descarregada gratuitamente a partir do site www.ubuntu.com

Os objetivos do projeto são, tal como enunciados pelo projeto [JIGSAW]:

- Criar um sistema de módulos de baixo nível para o Java.
- Ter como base o JSR-294.
- Aplicar esse sistema no JDK – Máquina virtual e plataforma base.
- Permitir a utilização deste sistema por terceiros e em projetos com âmbito distinto do JDK.

A SUN não pretende com este projeto, que o sistema modular Jigsaw faça parte de uma especificação Java aprovada pelo JCP, nem assegura que o sistema venha a ser suportado por outros fornecedores de ambientes e máquinas virtuais Java uma vez que o objetivo é dotar a sua versão de máquina virtual Java e plataforma Java de capacidades modulares [JIGSAW].

Assegura no entanto que, se e quando existir um sistema modular diferente na plataforma Java que faça parte de alguma especificação do JCP, providenciará as ferramentas necessárias para se poderem utilizar os módulos definidos no modelo Jigsaw nesse sistema de módulos [JIGSAW].

3.3 Definição do modelo Jigsaw

A especificação do modelo Jigsaw baseia-se, em muitos aspetos, na especificação definida pelo JSR 294 e também em algumas ideias da especificação JSR 277. Desta forma, no fim do projeto implementado, espera-se que a implementação possa representar uma implementação de referencia (IR[17]) para o JSR 294 [JIGSAW].

Tendo como base o JSR 294, a metodologia de módulos Jigsaw define que [JIGSAW]:

- Um módulo é um conjunto de classes e interfaces em um ou mais packages.

17 Em inglês, RI – Reference Implementation, pode traduzir-se para português como Implementação de Referência. Durante a dissertação será adotada a expressão inglesa porque encontra relação direta nas especificações.

- Os membros de um módulo são todas as classes e interfaces declarados em todas as unidades de compilação, designadas na definição por "*compilation units*", do módulo.

- Um módulo é identificado por uma estrutura do tipo **M@V**, sendo **M** o nome do módulo e **V** a versão do módulo.

- A unidade de compilação de um módulo é definida num ficheiro de código fonte com nome "module-info.java".

Tal como foi descrito na subsecção 2.2.5, o JSR 294 especifica um conjunto de alterações às linguagem e máquina virtual Java. O projeto Jigsaw, ao instanciar esses conceitos através de uma implementação real, provoca efetivamente alterações, necessárias para alcançar os objetivos pretendidos.

Essas alterações agrupam-se por:

- Alterações na gramática da linguagem Java

- Alterações no formato das classes que contêm código Java

- Alterações na máquina virtual Java

- Alterações na arquitetura de compiladores

- Introdução de novos comandos e conceitos

Estes aspetos são apresentados em detalhe nas subsecções seguintes.

Para efeitos de apresentação, este capítulo utiliza, simbolicamente, as letras **M**, **N** e **O** em representação de módulos, **T** para representar tipos (classes e interfaces), **C** em representação de unidades de compilação e **m** para representar meta-dados. Estas representações aparecem sempre a negrito e devidamente contextualizadas sendo utilizadas para melhorar a apresentação das várias definições. A sua utilização é também feita na definição original no projeto Jigsaw.

3.3.1 Gramática e regras semânticas

Para suportar módulos, tal como indicado no JSR 294, é necessário introduzir um conjunto de alterações à gramática da linguagem Java. O elemento principal a introduzir é o conceito de superpackage, já descrito anteriormente.

A gramática da linguagem Java, versão atual, é melhorada para suportar este conceito através de: i) alterações gramaticais à atual especificação de linguagem Java; ii) definição de regras semânticas, a aplicar na nova gramática definida.

3.3.1.1 Alterações gramaticais da especificação de linguagem Java

O Jigsaw define o conceito de superpackage presente no JSR 294 através da introdução do elemento gramatical "**module**" [JIGSAW].

A unidade gramatical "CompilationUnit", definida na especificação de linguagem Java, passa a considerar um novo tipo – *ModuleDeclaration*[18] – tal como indicado a seguir [JIGSAW]:

```
CompilationUnit:
  ImportDeclarations_opt ModuleDeclaration_opt PackageDeclaration_opt
ImportDeclarations_opt TypeDeclarations_opt
```

Tabela 3.3.1: Nova definição de elemento gramatical *CompilationUnit*

Adicionalmente, a gramática é estendida para definir o elemento *ModuleDeclation [JIGSAW]:*

```
ModuleDeclaration:
  Annotations_opt 'module' RestOfModuleDeclaration

RestOfModuleDeclaration:
  QualifiedIdentifier ';'
  ModuleId ModuleProvides_opt '{' { ModuleMetadata } '}'

ModuleProvides:
  'provides' ModuleId {',' ModuleId}_opt

ModuleMetadata:
  ModuleRequires
```

18 ModuleDeclaration, que em português pode traduzir-se por declaração de módulo. Nesta dissertação, o termo original será utilizado por referir diretamente um termo técnico de uma especificação.

```
  ModulePermits

ModuleRequires:
  'requires' {ModuleRequiresModifier}_opt ModuleId {',' ModuleId}_opt
';'

ModuleRequiresModifier:
  'optional'
  'private'
  'local'

ModulePermits:
 'permits' QualifiedIdentifier {',' QualifiedIdentifier}_opt ';'

ModuleId:
  QualifiedIdentifier ModuleVersion_opt

ModuleVersion:
  '@' Space_opt ModuleVersionStart ModuleVersionPart*
  '@' Space_opt StringLiteral

ModuleVersionStart:
  digit
  '['
  '('

ModuleVersionPart:
  InputCharacter but not FF, Space, Tab, '"', '\'', '\\', ',', ';'
  // InputCharacter is any UnicodeInputCharacter except CR and LF
```

Tabela 3.3.2: Gramática Java para suporte a módulos

A seguir é apresentado um exemplo compatível com a gramática definida.

```
module M1@1.0 provides M3@3.0 {
  requires jdk.base @ 7-ea;
  permits  M2;
}
```

Tabela 3.3.3: Exemplo de utilização de gramática para definição de módulo

O termo gramatical “modifier” é também alterado para passar a suportar o atributo “module”:

```
Modifier:
  Annotation
  'public'
  'module'
  'protected'
  'private'
  'static'
                                        ...
```

Tabela 3.3.4: Nova definição de elemento gramatical *Modifier*

A seguir é apresentado um exemplo compatível com a gramática de ”modifier”definida.

```
module int var1;
module String var2;
```

Tabela 3.3.5: Exemplo de utilização de gramática do termo “modifier”

Com estas alterações simples, a nova gramática da especificação de linguagem Java suporta:

- o conceito superpackage designado no JSR -294, dando para o efeito o nome de “module”.
- a existência de módulos com versões associadas, embora não atribua significado especial às versões.

3.3.1.2 Conceitos semânticos introduzidos

Um conjunto de conceitos semânticos é também introduzido na especificação de linguagem Java através do Jigsaw que se aplicam na gramática definida.

Estes conceitos têm como objetivo: i) definir de que forma um módulo exporta os tipos; ii) definir de que forma um módulo declara as suas dependências com outros módulos; iii) definir de que forma se declara um módulo; iv) definir de que forma as atuais formas de declaração de unidades de compilação existem com as novas unidades de compilação associadas à definição de módulos.

3.3.1.2.1 Definição de módulo

Um módulo defini-se da seguinte forma [JIGSAW]:

- A unidade de compilação de um módulo é um ficheiro com nome "module-info.java".
- É na unidade de compilação de um módulo que é definido o nome do módulo relativamente ao qual os tipos pertencem.
- O termo "module" é um termo restrito na gramática, não podendo ser utilizado noutro contexto que não no da própria definição de módulo.
- O nome de um módulo pode ser simples ou qualificado, isto é, um nome composto por vários nomes simples unidos por o símbolo ".". O nome de um módulo não apresenta qualquer restrição relativamente a outros nomes, podendo por exemplo existir um módulo com o mesmo nome de um package.

3.3.1.2.2 Dependências e relações entre módulos

A declaração de um módulo, numa unidade de compilação de módulo, pode conter meta-dados que permitam indicar:

- As dependências perante outros módulos.
- Outros nomes pelos quais o módulo pode ser identificado.
- De que forma outros módulos podem requerer dependências relativamente a este módulo.

Trata-se de uma das características mais importantes numa metodologia modular. Permite dar sentido à estrutura, isto é, fornecer a informação de que forma é que todos os blocos estruturados se relacionam entre si.

O Jigsaw define, relativamente a este aspeto, um conjunto de definições importantes [JIGSAW]:

- Quando a declaração de um módulo **M** indica o atributo "permits" na sua definição, apenas os módulos indicados por este atributo podem declarar o módulo **M** como dependência. Caso a declaração de um módulo **M** não indique o atributo "permits", qualquer módulo pode declarar dependência do módulo **M**. Esta regra permite definir quem pode ou não utilizar um determinado módulo.

- Quando a declaração de um módulo **M** indica o atributo "requires" na sua definição, a ambiente Java necessita disponibilizar ao módulo o conjunto de módulos indicados, nas versões indicadas. Basicamente, é desta forma que, ao desenhar-se um módulo, se define que módulos vão ser necessários, indicando desta forma ao compilador e à máquina virtual a necessidade de estes garantirem que os módulos indicados estão presentes e visíveis para o módulo M.

- A linguagem Java não impõe nenhuma ordem na declaração das dependências de um módulo **M**. Se um módulo **M** depende de um módulo **N** que contêm um tipo **T** e, o módulo **M** também depende de um módulo **O** que reexporta o mesmo tipo **T**, compete ao ambiente Java utilizar **N** ou **O** como módulo a utilizar para disponibilização de visibilidade do tipo **T**. Significa que um tipo **T**, pode estar visível através de 2 dependências distintas, **N** e **O** e que o ambiente pode optar por utilizar o tipo **T** visível através de **O** ou utilizar o tipo **T** visível através de **N**, sem que esta ordem esteja relacionada com a ordem de declaração de dependências na definição do módulo **M**.

- O atributo "optional", na declaração de dependência de um módulo instrue o ambiente a não dar erro algum se for tentado utilizar o módulo declarado na dependência, independentemente de estar ou não presente no ambiente.

- O atributo "local", utilizado para indicar a dependência de um módulo **M** relativamente a um módulo **N**, instrue o ambiente a utilizar o mesmo ClassLoader para carregar todos os tipos dos dois módulos, **M** e **N**. Se o atributo não estiver presente, o ambiente pode optar por utilizar diferentes ClassLoaders para diferentes módulos.

- O elemento "provides", na declaração de um módulo **M**, indica que, todos

os módulos, **N1**, **N2**, …, que dependam dos módulos, **O1**, **O2**, …, indicados neste atributo, podem ver as suas dependências satisfeitas através da utilização do módulo **M**.

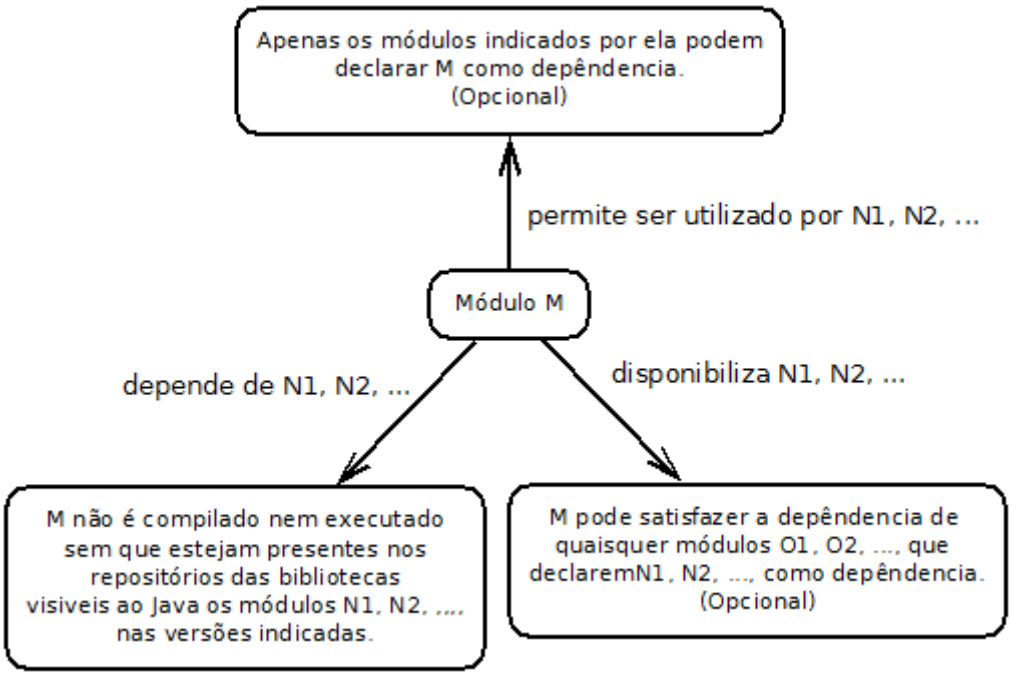


Figura 3.3.1: Relação de dependências de um módulo

Por exemplo, são definidos dois módulos, **M1** e **M2**. **M2** depende de **M1** e da plataforma base, **jdk.base @ 7-ea**. **M1** depende da plataforma base **jdk.base @ 7-ea**

```
module M1@1.0 provides M3@3.0 {
  requires jdk.base @ 7-ea;
  permits M2;
}
```

Tabela 3.3.6: Ficheiro *module-info.java* para definição do módulo M1.

Para definição do módulo **M1**, a unidade de compilação de módulo **M1**, com versão **1.0**, declara dependências com os módulos **jdk.base**, versão **7-ea** e, adicionalmente, apenas permite ser declarado como dependência ao módulo **M2.**

```
module M2@1.0 {
  requires jdk.base @ 7-ea, private M1 @ 1.0;
  permits M3;
}
```

Tabela 3.3.7: Ficheiro *module-info.java* para definição do módulo M2.

Para definição do módulo **M2**, a unidade de compilação de módulo **M2**, com versão **1.0**, declara dependências com os módulos **jdk.base**, versão **7-ea** e com o módulo **M1**, versão **1.0** e, adicionalmente, apenas permite ser declarado como dependência ao módulo **M3**.

Todos os tipos que **M1** exporta não são reexportados por **M2** para o módulo **M3** uma vez que **M2** depende de **M1** de forma privada.

3.3.1.2.3 Acessibilidade com módulos

A acessibilidade define de que forma os tipos (classes, interfaces) são acessíveis, ou visíveis, perante outros tipos.

Atualmente, pode-se controlar a visibilidade de tipos em 4 camadas: i) classe; ii) classe e sub-classes; iii) package; iv) visível para todos as classes.

Os módulos introduzem uma nova camada de controlo de visibilidade: modulo. Passamos a ter os seguintes níveis: i) classe; ii) classe e sub-classes; iii) package; iv) módulo; v) visível para todos as classes e todos os módulos.

Para além destas camadas, as dependências entre módulos também estão classificadas do ponto de vista de visibilidade, isto é, as relações entre módulos podem ser marcadas com atributos que controlam de que forma os tipos são visíveis entre as dependências.

O Jigsaw define [JIGSAW]:

- Se uma classe ou interface é declarada como "public", pode ser acedida a partir de qualquer código, assumindo que a unidade de compilação onde é declarada está visível para esse código.

- Se uma classe ou interface não é declarada “public” nem “module”, pode ser apenas acedida no package onde está declarada.

- Se uma classe ou interface é declarada “module”, pode ser acedida apenas no módulo onde está declarada.

- Todos os tipos declarados públicos num módulo são visíveis para todos os tipos no módulo e em todos os módulos que dependem deste módulo. O ambiente tem que utilizar as dependências de cada um dos módulos para determinar que tipos declarados públicos fora do módulo são visíveis para os tipos no módulo. Se um tipo **T** é declarado publico e o tipo **T** não está num módulo **M**, o tipo **T** é visível para qualquer tipo declarado em **M** se se verificam cumulativamente as 3 condições seguintes:

  – o ambiente consegue encontrar um módulo, **N**, que contêm ou reexporta **T** e,

  – o módulo **M** depende do módulo **N** e,

  – o módulo **N** permite que o módulo **M** declare dependência do módulo **N**.

- Um módulo **M** reexporta um tipo **T** se:

  – o tipo **T** não é um tipo que pertence ao módulo **M** e que qualquer tipo em **M** consegue observar o tipo **T** e,

  – a dependência do módulo **M** relativamente ao módulo onde **T** pode ser observado não está marcada como privada “private” nos meta-dados do módulo **M**.

- Se um tipo **T** pertence a um módulo **M** e esse tipo está também visível a partir de um outro módulo relativamente ao qual **M** também depende, a linguagem Java não impõe qualquer definição de preferência sobre qual o tipo **T** que deve ser utilizado. No entanto, o ambiente deve escolher o tipo **T** de um dos módulos e utilizar esse tipo no âmbito do módulo **M** de forma consistente e permanente.

- Um membro (classe, interface, variável ou método) de uma referência a um tipo Java (classe, interface ou *array*) ou a um construtor de uma classe é

acessível apenas se o tipo é acessível e o membro ou o construtor são declarados para permitir acesso da seguinte forma: i) Se o membro ou o construtor está declarado com atributo "*public*", o acesso é permitido ou; ii) Se o membro ou construtor é declaro como um "module", o acesso é permitido apenas no módulo que contem a classe no qual o membro ou construtor está declaro.

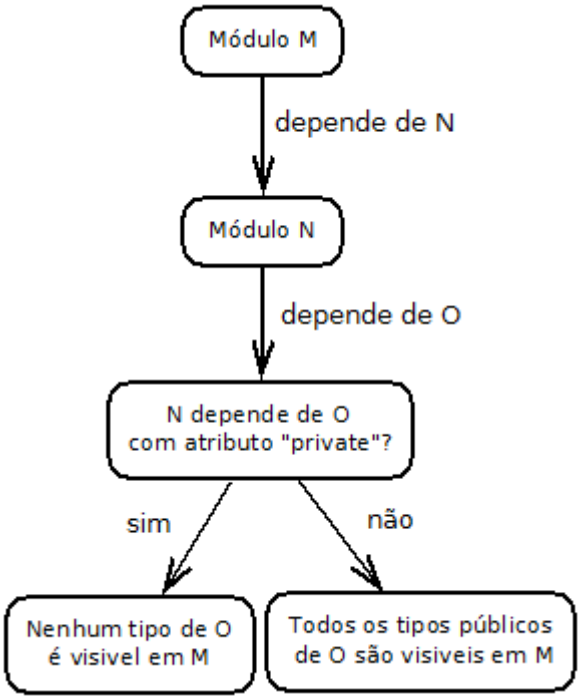


Figura 3.3.2: Visibilidade de tipos entre módulos

3.3.1.2.4 Anotações

A funcionalidade da linguagem Java – Anotações – pode ser utilizada na declaração de um módulo. Através desta funcionalidade podemos definir atributos que melhor caracterizam a definição do módulo, permitindo guardar diretamente na definição do módulo meta-informação relevante para a sua caracterização.

O Jigsaw define as seguintes regras na utilização desta funcionalidade com a declaração de módulos [JIGSAW]:

- Um módulo pode ter no máximo uma anotação.

- Para definir uma anotação para um módulo, esta anotação deve ser colocada na unidade de compilação do módulo, que, quando em sistema de ficheiros, corresponde a um ficheiro com nome "module-info.java".

- Para definir a anotação, o termo gramatical "module" na unidade de compilação do módulo é precedido de termos gramaticais relativos à funcionalidade de anotação.

Não estão ainda definidas o conjunto de anotações consideradas úteis, nomeadamente para o compilador. Fica apenas em aberto a possibilidade de virem a ser utilizadas no futuro.

3.3.2 Versões

Um módulo fica definido por um nome e por uma versão. Dentro de uma biblioteca, podem existir módulos com o mesmo nome desde que as versões sejam diferentes. Não podem existir módulos com o mesmo nome e a mesma versão numa mesma biblioteca [JIGSAW].

A designação lexical para a versão é ampla e não tem nenhuma relação direta com alguns dos sistemas tradicionais que ligam as versões a séries numéricas. Podemos ter versões que não contêm dígitos alfanuméricos.

São exemplos de versões corretamente definidas: "**a1**"; "**1.0**"; "**1.0beta**"; "**a**";

Esta abrangência de sintaxe tem como consequência direta a forma como as dependências entre módulos são declaradas. A definição das dependências de um determinado módulo pode ser realizada de 2 formas:

- Declarando a dependência relativamente a um módulo não especificando qual a versão pretendida – Nesta situação a máquina virtual vai satisfazer a dependência através de qualquer módulo que satisfaça a dependência, presente nas bibliotecas visíveis, em qualquer versão encontrada.

- Indicando qual o módulo pretendido e a versão exata do módulo.

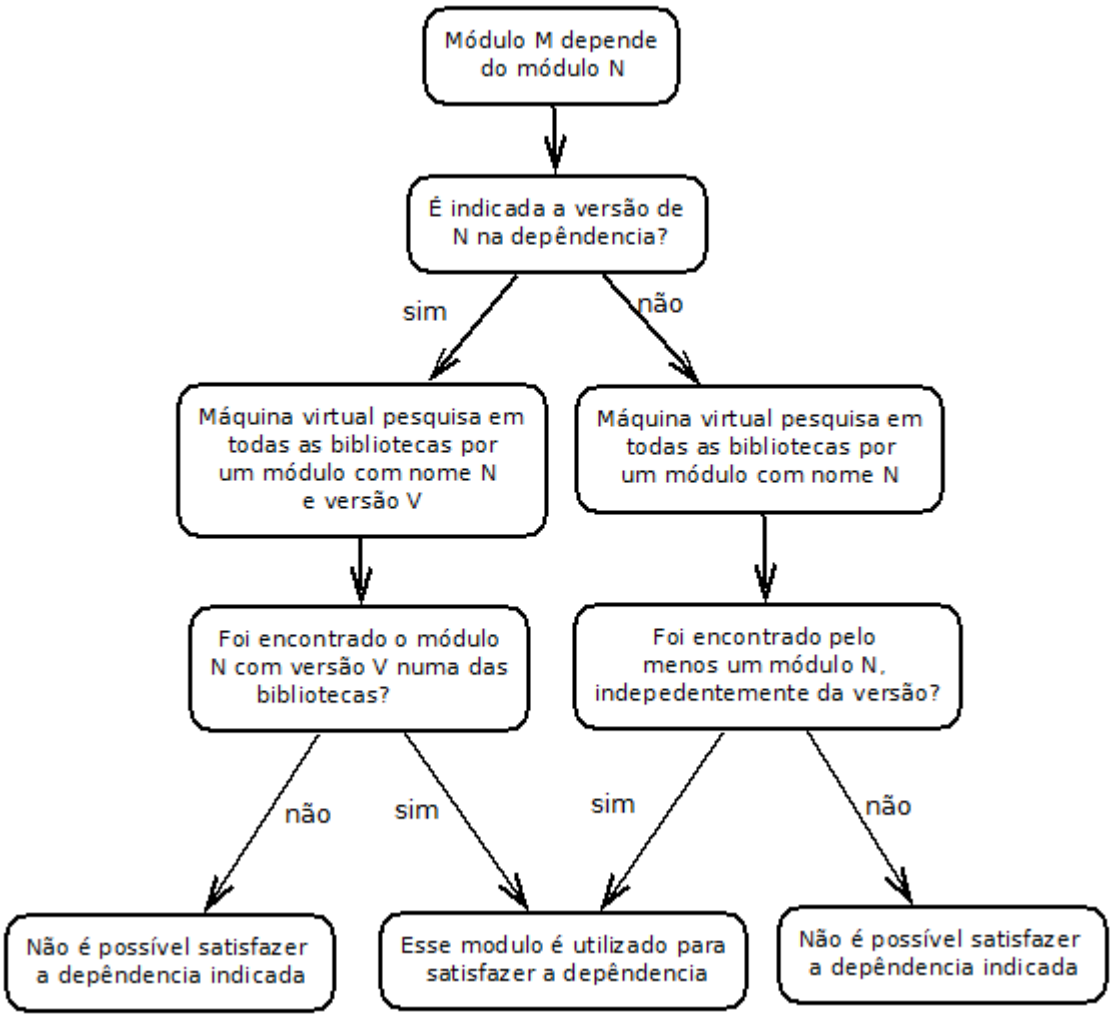


Figura 3.3.3: Método de resolução de versões de módulos

3.3.3 Bibliotecas de módulos

Está definido, no âmbito do projeto, que os módulos são arquivados em repositórios. Estes repositórios representam a localização e armazenamento dos módulos. Os repositórios estarão agregados em bibliotecas de módulos – Library na definição – que serão na verdade o ponto de contacto e de acesso dos módulos do ponto de vista funcional.

Esses repositórios terão como requisitos base [JIGSAW]:

- Capacidade de armazenamento de múltiplas versões de módulos.

- Capacidades de acessos partilhados e concorrentes.
- Uma arquitetura corretamente definida através de uma metodologia exata, através da existência de uma API.

Não é exigido que os repositórios estejam armazenados localmente.

As bibliotecas definem as seguintes capacidades base [JIGSAW]:

- Gerir o conteúdo dos vários repositórios.
- Permitir uma gestão otimizada dos repositórios locais e remotos.
- Permitir satisfazer as dependências de módulos procurando nos repositórios as versões necessárias e instanciando essas versões no ambiente Java.
- Permitir uma relação de dependência hierárquica com outras bibliotecas e permitir satisfazer a necessidades de módulos nessas bibliotecas hierarquicamente relacionadas quando não é possível satisfazer localmente.
- Na relação de dependências com outras bibliotecas, a especificação atual, ainda em processo de revisão, define que uma biblioteca pode ter nenhuma ou no máximo uma biblioteca que é definida como “parent”, isto é, biblioteca pai. No caso de uma biblioteca não definir nenhuma biblioteca pai, essa biblioteca têm implicitamente como biblioteca pai a biblioteca da plataforma. Desta forma, a biblioteca da plataforma Java é a biblioteca pai de todas as outras bibliotecas existentes.

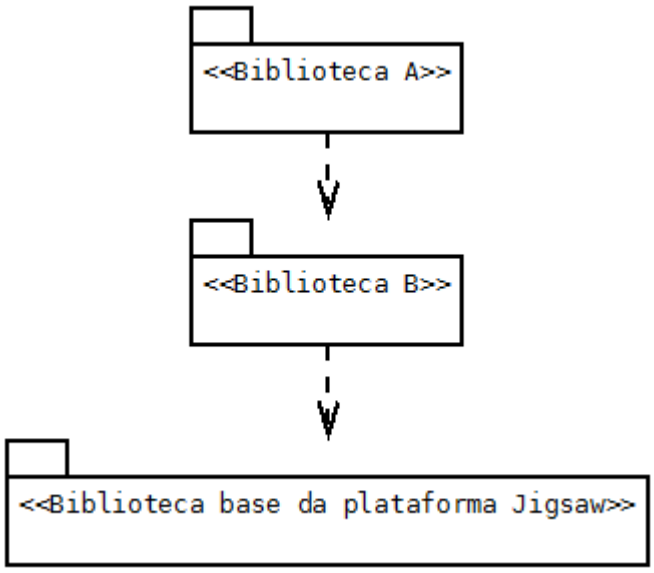


Figura 3.3.4: Modelo de bibliotecas Jigsaw

Atualmente, o modelo de referência que está desenvolvido define e representa as bibliotecas e repositórios como diretorias no sistema de ficheiros, com ficheiros especiais que caracterizam e descrevem o conteúdo dessas diretorias, estando as respetivos versões de módulos nessas diretorias, armazenadas em sub-diretorias. Esta não é no entanto uma limitação, podendo os repositórios estar em bases de dados, servidores remotos ou outros locais que permitam o acesso programaticamente.

A plataforma JDK vai ter, segundo o que está especificado, um repositório base para armazenamento de todos os módulos da plataforma base, que deverá ser partilhado por todo o sistema, nomeadamente por todas as versões de plataforma Java instaladas.

Para permitir a gestão de bibliotecas de módulos, foi desenvolvida uma ferramenta, parte dos utilitários do JDK, chamada *jmod*.

3.3.4 Máquina virtual

São necessárias alterações na máquina virtual Java, por forma a suportar a metodologia de modularidade Jigsaw, que, ao ter como um dos objetivos suporte de modularidade na própria máquina e plataforma base, exige obrigatoriamente

que a especificação de máquina virtual sofra alterações.

As alterações introduzidas visam garantir as regras semânticas definidas em 3.3.1.

As alterações fazem-se nos seguintes contextos [JIGSAW]:

- processo de carregamento e resolução de classes
- controlo de acesso

As alterações pormenorizadas a realizar são realizadas no código fonte do compilador Java e são de muito baixo nível, não estando no âmbito desta dissertação o seu estudo pormenorizado.

3.3.5 Ficheiros de classes

Os ficheiros de classes têm que ser estendidos para passar a suportar a existência de módulos.

As alterações contemplam a criação de novas definições de dados, a adição de novos atributos, a definição da estrutura para o ficheiro correspondente à unidade de compilação de módulo – module-info.java – entre outras.

A versão das classes com suporte a módulos será a 51 e não será suportada por versões Java anteriores à versão Jigsaw.

As alterações a realizar são incrementais mas extensas e de muito baixo nível e não são detalhadamente abordadas no âmbito desta dissertação.

3.3.6 Arquitetura dos compiladores

O compilador Java, que tem o nome *javac* no JDK, é também estendido para suportar módulos. Durante a escrita desta dissertação, as alterações ainda não estão completamente fechadas no projeto Jigsaw pelo que até à versão final, podem sofrer correções.

3.3.6.1 Requisitos

Um conjunto de requisitos está definido pelo projeto relativamente aos compiladores.

Esses requisitos são [JIGSAW]:

- o compilador atual, no JDK, **javac,** necessita continuar a funcionar de forma transparente com os atuais processos de compilação, incluindo IDEs, eventualmente com pequenas alterações na sequência de compilação.
- Deve ser possível utilizar classes em módulos sem ter que as instalar em bibliotecas de módulos e deve ser possível realizar compilações incrementais.
- Deve ser possível compilar utilizando ficheiros de código fonte ou classes compiladas de módulos instalados na biblioteca de módulos do sistema.
- Deve ser possível indicar uma política de resolução de qual a versão de módulo a utilizar quando mais do que uma versão está disponível nomeadamente utilizar a mais antiga ou utilizar a mais recente.

3.3.6.2 Ficheiros e formatos

Seguindo os processos formais já estabelecidos na especificação de máquina virtual Java, um conjunto de melhorias é introduzido para suportar módulos:

- Tal como acontece com a definição de packages, recorrendo aos ficheiros "package-info.java", os meta-dados de módulos devem ser escritos, na versão código fonte, em ficheiros com nome "module-info.java".

3.3.6.3 Detalhes de implementação

As maiores alterações ao compilador são na forma como ele acede classes que não são indicadas nos parâmetros de linha de comandos.

As alterações a realizar são incrementais mas extensas e de muito baixo nível e não são detalhadamente abordadas no âmbito desta dissertação.

3.3.7 Alterações noutras ferramentas do ambiente JDK

Está definido, no âmbito do projeto, que outras ferramentas típicas do ambiente de desenvolvimento Java – JDK, tais como ***javadoc***[19] e ***javap***[20], vão necessitar de ser atualizadas e melhoradas para passar a considerar a existência de módulos, nomeadamente para passarem a ter visibilidade sobre a biblioteca de módulos do ambiente. Estas alterações não estão ainda completamente identificadas no âmbito do projeto Jigsaw. Outras ferramentas novas vão ser desenvolvidas, nomeadamente para gerir a instalação e manutenção de módulos.

3.4 Compilação e execução

A subsecção 3.4 apresenta os processos associados ao desenvolvimento de aplicações em Jigsaw, descrevendo os cenários principais de utilização da plataforma Java no desenvolvimento Java.

Atualmente, com a compilação de programas Java, todos os tipos necessários têm que estar presentes no ambiente para que a compilação tenha sucesso. Com o Jigsaw, essa validação é estendida aos módulos, exigindo que todos os módulos necessários estejam presentes no ato da compilação.

O Jigsaw prevê 3 modelos de resolução de tipos [JIGSAW]: Nativo, tradicional e híbrido.

Modo nativo – neste modo, as aplicações são compiladas e executadas utilizando todos os tipos (classes, interfaces, …) presentes em todos os módulos resultado das dependências da aplicação, através do mecanismo nativo de resolução e carregamento de classes Jigsaw, com todas as regras semânticas definidas pelo Jigsaw. Desta forma, a utilização de caminhos de classes, ficheiros JAR e outros métodos tradicionais deixa de ser utilizado e todas as dependências são resolvidas exclusivamente recorrendo às bibliotecas de módulos disponíveis. Neste modo, todo o código não definido no âmbito de módulos fica inutilizado.

19 javadoc – ferramenta da plataforma Java para geração automática de documentação de API.

20 javap – ferramenta da plataforma Java para análise de performance e deteção de bugs durante a execução.

Modo tradicional – neste modo, as aplicações são compiladas e executadas utilizando os processos de descoberta, resolução e carregamento de classes existentes nas versões atuais do Java: caminho de classes, ficheiros JAR ou outros mecanismos de armazenamento de classes. Este modo representa compatibilidade com a atual versão J2SE. Apesar de o compilador e máquina virtual funcionarem com compatibilidade com o J2EE, versão 6, existem no entanto aplicações que deixarão de funcionar. Essas aplicações são aquelas que, por questões várias resultantes do desenho funcional, tomaram assunções relativamente à hierarquia de ficheiros e diretorias onde reside a plataforma Java. O modo tradicional assegura compatibilidade de funcionamento não podendo no entanto assegurar compatibilidade da hierarquia de ficheiros e diretorias base da plataforma Java, uma vez que a plataforma Java se encontra modularizada apresentado um novo modelo de diretorias e repositório de código. Essas aplicações terão obrigatoriamente que ser alteradas uma vez que deixaram de funcionar adequadamente, com efeitos que podem ter impacto suficiente para as tornar inutilizáveis.

Modo híbrido – neste modelo, é possível compilar e executar aplicações modulares e permitir que estas acedam a packages não modularizados. Este modelo permitirá uma transição mais simples do atual modelo para o novo modelo. Trata-se de uma abordagem que está a ser discutida no âmbito do projeto para facilitar a reutilização do código já existente não sendo no entanto uma prioridade do projeto e não estando garantida a sua implementação uma vez que o projeto Jigsaw não depende deste modo para executar corretamente. A figura seguinte apresenta o processo de funcionamento do modo híbrido.

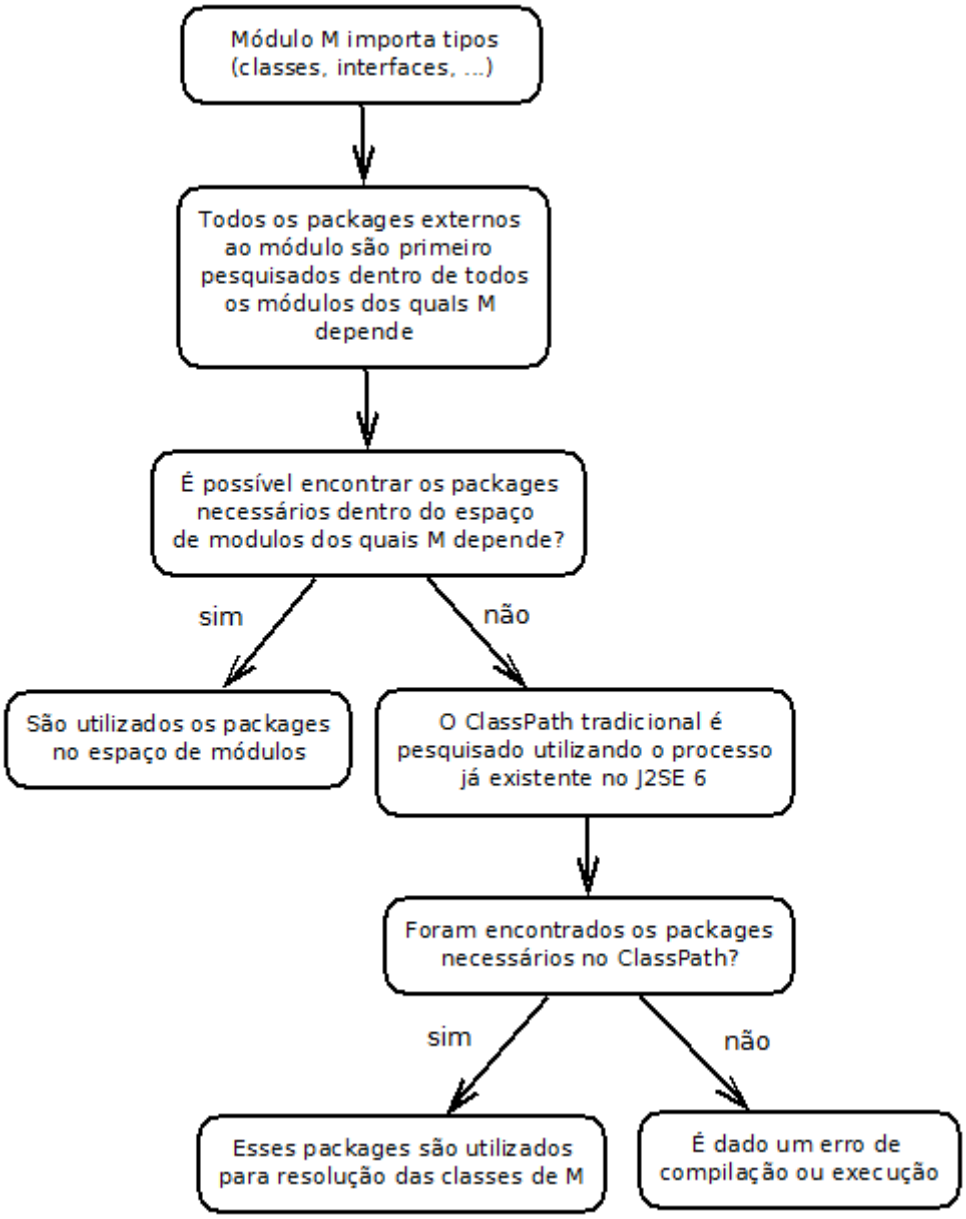


Figura 3.4.1: Método de resolução e carregamento de classes no modo híbrido

Apesar de estes modos serem os modos que estão previstos, existe ainda alguma indefinição de como implementar estes modos, nomeadamente o modo híbrido, não existindo a garantia que este modo de execução venha a ser implementado.

3.5 Integração com o sistema operativo

É objetivo do projeto do projeto permitir uma melhor integração da plataforma JDK com o sistema operativo ao nível de gestão de packages Java.

Atualmente, a plataforma Java, com as suas bibliotecas base, não está integrada no sistema operativo como qualquer outra aplicação tipicamente está. A plataforma Java é, tipicamente instalada como uma aplicação que, depois de forma privada, faz a gestão de todos os packages Java instalados, não estando a arquitetura de gestão desses packages orientados ao sistema operativo mas sim à plataforma Java onde eles residem.

Desta forma, é possível ter múltiplas instalações paralelas, de diferentes versões, da plataforma Java, com duplicação de packages, que muitas das vezes representam a mesma versão. A gestão deste processo é exigente e propensa a criação de erros de utilização, como por exemplo a utilização da versão incorreta da plataforma.

O projeto Jigsaw endereça esta situação, ao prever uma integração mais profunda com o sistema operativo onde está instalado e com as capacidades de gestão de software deste.

Estas alterações possibilitam: i) a existência de múltiplas instalações da plataforma Java, apenas com os packages Java essenciais; ii) a existência de um repositório central ao sistema operativo, de módulos Java, devidamente gerido com versões, acessível a todas as versões de plataforma Java instaladas. iii) a capacidade de cada uma das plataformas utilizar esse repositório para satisfazer as interdependências de módulos, recorrendo desta forma a um repositório central comum e único; iv) a possibilidade de integrar a gestão desse repositório com ferramentas de gestão do sistema operativo, nomeadamente ferramentas de instalação e atualização de software.

Esta abordagem tem como objetivo facilitar a gestão de múltiplos módulos, com múltiplas versões, num sistema complexo, sem possibilidade de sobreposição ou qualquer outro erro típico da gestão de packages de software em sistemas.

Assim prevê-se que a instalação e gestão da plataforma Java possa ser realizada pelas próprias ferramentas dos sistemas operativos, como por exemplo [JIGSAW]:

- RPM[21], nas versões Linux que suportam o sistema de gestão de packages de software RPM.
- DEB[22], nas versões Linux que suportam o sistema de gestão de packages DEB.
- Windows *Installer*, nos sistemas operativos Windows.

Nada está ainda definido nem fará parte de alguma especificação, tratando-se de algo que o Jigsaw considera como boas práticas.

A titulo de exemplo, a versão Jigsaw atualmente existente, demonstra as intenções do projeto, ao integrar de o processo de instalação do JDK no sistema de gestão nativo do Ubuntu.

3.6 Comparação com outras metodologias

Estudada em detalhe a metodologia Jigsaw, interessa compará-la com as metodologias apresentadas no Capítulo 2.

Na tabela seguinte é apresentada a comparação já apresentada no Capítulo 2, agora incluindo o Jigsaw. É importante para perceber como se compara, de forma direta, com as metodologias apresentadas.

21 RPM, sigla inglesa da expressão **R**edHat **P**ackage **M**anager, que em português significa Gestor de Pacotes RedHat

22 DEB, representa uma forma de gestão e arquivo de pacotes de software em certas distribuições Linux, nomeadamente o Ubuntu.

	JSR 277	JSR 291	JSR 294	AOP	Jigsaw
Abstração de implementação e de dados	S	S	S	S	S
Conceito de módulo	S	S	S	S	S
Conceito modular na especificação Java			S		S
Conceito modular na máquina virtual Java			S		S
Políticas de segurança aplicáveis aos módulos na especificação Java			S		S
Repositório de módulos	S	S			S
Características de gestão de versões	S	S			S
Define todos os sistemas necessários à implementação de um sistema modular	S	S			S
Existe implementação de referência	S	S			S
JSR aprovado pelo JCP		S	S		

Tabela 3.6.1: Tabela comparativa da JSR 277, JSR 291, JSR 294, AOP e Jigsaw

A letra "S" significa, na tabela a existência da propriedade referenciada.

Conclui-se que o Jigsaw consegue, de forma simples como se demonstrou, corresponder a todas os aspetos que caracterizam aquilo que foi definido como as bases para uma metodologia modular completa. Um aspeto que torna o Jigsaw diferente de todas as outras metodologias é que, com a abordagem de alterar a máquina virtual Java e linguagem Java, consegue integrar a modularização no Java de uma forma simples e fazendo com que modularidade seja algo quase transparente para os programadores, tal como foi desde o início a gestão automática de memória por parte da máquina virtual Java[23]. A modularidade está lá mas os programadores não necessitam de se preocupar com os aspetos técnicos que a suportam, apenas com os aspetos funcionais.

23 No Java, a gestão de memória por parte das aplicações (alocação e libertação) é um processo que é assegurado de forma transparente pela máquina virtual Java, deixando o programador concentrado nos aspetos funcionais da aplicação.

Da análise das alterações propostas, tanto na máquina virtual como na linguagem, verifica-se a garantia de retro-compatibilidade do ponto de vista formal na linguagem Java, isto é, as alterações introduzidas não alteram nada na gramática já existente nem nas regras semânticas existentes, introduzindo apenas melhorias incrementais que se traduzem numa nova unidade de compilação – definição de módulo.

3.7 Conclusões

O projeto Jigsaw propõe, como principal objetivo, modularizar a plataforma Java, na próxima versão, Java 7, através da definição de um sistema elementar de modularidade, complementado por uma lógica de gestão de uma plataforma modular, que está disponível na máquina virtual Java, na plataforma base Java e todas as APIs base: i) java.lang.*; ii) java.util.*; …; e, que pode, opcionalmente, ser utilizado no desenvolvimento de sistemas aplicacionais escritos em Java, com utilizações mais ou menos avançadas conforme o âmbito que se pretender, uma vez que, apesar da especificação introduzir alterações, é completamente compatível com o atual código.

Com o projeto Jigsaw obtemos uma plataforma Java modular. Percebemos também que, embora não seja o objetivo principal do projeto Jigsaw, ficamos também com um conjunto de facilidades disponíveis na linguagem Java, na máquina virtual Java e nas bibliotecas que constituem a plataforma Java para implementar aplicações Java modulares.

Importa agora demonstrar como utilizar este sistema no desenvolvimento de aplicações e as vantagens e desvantagens de adotar este sistema de modularização relativamente a outros já existentes, nos vários cenários, nomeadamente no desenvolvimento de projetos software para sistemas de servidores aplicacionais.

Capítulo 4

Desenvolvimento modular em Jigsaw

4.1 Introdução

No Capítulo 3 é apresentada a metodologia Jigsaw, que permite implementar toda a plataforma Java de forma completamente modular e em paralelo, dotar a linguagem Java de suporte a desenvolvimento modular.

Este capítulo apresenta o estudo de aplicabilidade da metodologia Jigsaw no desenvolvimento de aplicações Java modulares, focando o estudo na aplicabilidade da metodologia no desenvolvimento e integração de servidores aplicacionais Java.

Este capítulo divide-se em 2 fases distintas. A primeira foca-se no estudo sobre a utilização do Jigsaw para desenvolvimento de aplicações Java de uma forma geral. A segunda foca o estudo no desenvolvimento e integração com servidores aplicacionais e de que forma as metodologias de modularidade, como por exemplo plugins, podem ser substituídas pela proposta pelo Jigsaw.

4.2 Conceitos

Importa agora definir um conjunto de conceitos e definições, para melhor definir o âmbito do estudo, balizando objetos importantes do estudo.

Tal como é apresentado, o estudo é sobre desenvolvimento modular de

aplicações em Java, focando depois a atenção em servidores aplicacionais.

As arquiteturas de servidores Java evoluíram ao longo dos últimos anos, de um modelo cliente servidor, ainda hoje utilizado, por exemplo em servidores de base de dados SQL, como é o caso do servidor MySQL [MYSQL 10], primeiro para servidores web, com camadas aplicacionais específicas e orientadas para geração dinâmica de conteúdos HTML para a internet, como por exemplo o servidor Tomcat [TOMCAT 10] e, numa segunda fase, para uma arquitetura mais complexa, orientada para plataformas aplicacionais de prestação de serviço, com base em arquiteturas multi-nível, como é o caso de todos os servidores que implementam a especificação JEE.

No âmbito desta dissertação, e de forma a diferenciarmos aplicações Java de um modo geral das aplicações que representam servidores aplicacionais Java, definimos que um um servidor aplicacional Java consiste numa aplicação, com arquitetura multi-nível, com várias camadas que se organizam do ponto de vista funcional, que disponibilizam uma infraestrutura na forma de serviços, e possibilitam a construção e execução de aplicações que resolvam de forma eficiente problemas de várias áreas, utilizando como base a definição de servidor aplicacional JEE [Kassem 00].

Consideram-se neste grupo servidores comerciais tais como Resin, da empresa Caucho, OC4J, da empresa Oracle, GlassFish Enterprise Server, da empresa Oracle (um servidor que incorpora tecnologia modular OSGi), WebLogic Server, da empresa Oracle, SAP Netweaver AS, da empresa SAP, JBoss da empresa RedHat, WebSphere Application Server da empresa IBM, Apache Geronimo, da Apache Software Fundation. Estes servidores representam atualmente uma quota importante do mercado de servidores aplicacionais Java.

O que caracteriza estes servidores é o facto de todos eles implementarem a especificação JEE, sendo que os servidores JEE representam a maior cota de mercado de servidores Java.

Os servidores JEE são cacterizados por implementar uma arquitetura multi-camada e tipicamente estão organizados em 3 camadas [Kassem 00]: i) camada de apresentação – responsável pela camada de apresentação, nomeadamente a

geração de conteudos; ii) camada lógica de negócio – responsável pelas funcionalidades propriamente ditas que resolvem os problemas especificos que se pretendem ver resolvidos; iii) camada de dados – que garante um conjunto de serviços que assegura o arquivo e pesquisa de dados de forma escalável, eficiente e segura.

Adicionalmente, existe toda uma infra-estrutura funcional necessária para suportar todas as 3 camadas e a ligação entre elas.

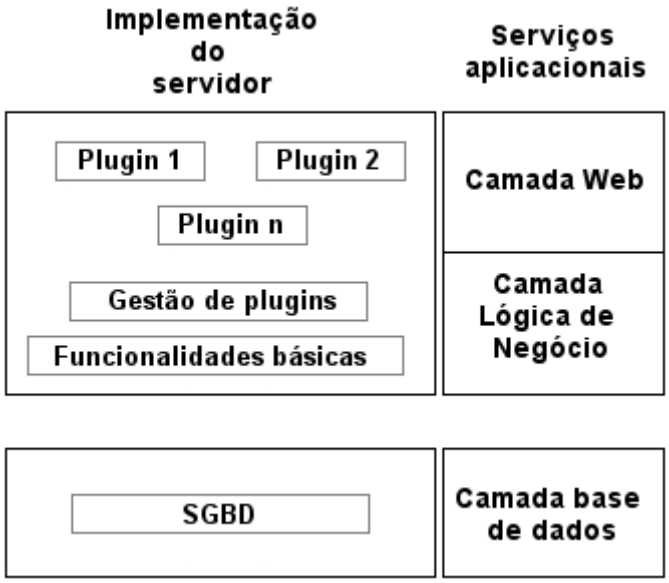


Figura 4.2.1: Arquitetura tradicional de um servidor JEE

É com base na arquirectura tipica dos servidores JEE que procederemos ao estudo de integração e deploy da metodologia Jigsaw.

4.3 Desenvolvimento de aplicações Java com o Jigsaw

O estudo inicia-se estudando a forma como a metodologia Jigsaw pode ser aplicada ao desenvolvimento de aplicações Java de forma geral, não incluindo neste grupo servidores aplicacionais.

Para fundamentar o estudo, observou-se os métodos adotados pela equipa

que está a desenvolver o projeto Jigsaw e, verificam-se os seguintes aspetos necessários que estão em prática no desenvolvimento da plataforma Jigsaw, ela própria a ser desenvolvida segundo a metodologia modular.

Da análise efetuada e tendo como ponto de partida o código fonte do projeto Jigsaw, verifica-se que é necessário [JIGSAW]:

- Identificar corretamente os módulos que constituem o projeto.
- Identificar/classificar todos os tipos (classes e interfaces) e packages que pertencem a cada um dos módulos.
- Definir as relações de dependências entre os módulos.

Uma vez identificados os pontos enumerados atrás, o desenvolvimento da aplicação decorre como se um projeto não modular se tratasse.

Num modelo em que não se utiliza nenhuma metodologia modular para além de ficheiros JAR, um projeto de software é desenvolvido tendo como base código fonte, distribuído por packages, e um conjunto de tipos Java (classes, interfaces), acessível através do *Classpath*[24]. Na plataforma Java, Classpath representa um conjunto de caminhos no sistema de ficheiros onde a máquina virtual Java e o compilador Java devem pesquisar sempre que necessitam de satisfazer dependências de packages e tipos Java (classes, interfaces) e é utilizado pelos ClassLoaders da plataforma.

No modelo de desenvolvimento Jigsaw, as únicas alterações necessárias são externas ao código Java e são acima de tudo formais, podendo enumerar-se por:

- definir formalmente os módulos, criando para cada um deles os respetivos ficheiros de suporte “module-info.java”.
- alterar os processos de compilação para passar a compilar todo o código fonte, de forma organizada por módulos, incluindo em cada um dos módulos o ficheiro respetivo “module-info.java”.

24 Classpath pode traduzir-se em português por caminho de classes. Classpath é definido no âmbito da plataforma Java e será utilizado por melhor definir o sentido original da definição.

A simplicidade do modelo é que, o código fonte dos packages não sofre qualquer alteração, sendo exatamente o mesmo com ou sem módulos, sendo necessário apenas definir os módulos através da escrita de ficheiros "module-info.java" para cada um dos módulos, e, no ato da compilação, atribuir cada um dos tipos em código fonte ao módulo respetivo.

Após compilados, os tipos ficam, nos ficheiros de classes, definidos com o módulo a que pertencem. É nesta altura que existem as primeiras alterações não retro-compatíveis, uma vez que as classes compiladas deixam de poder ser executáveis pelas plataformas Java anteriores ao Jigsaw. Os módulos são instalados numa biblioteca e executados de acordo com o processos definidos no Jigsaw.

No processo de execução, os módulos necessários à execução da aplicação são carregados automaticamente e de forma transparente à aplicação, não sendo necessário qualquer programação especifica para suportar módulos.

A simplicidade pode ser demonstrada, partindo de um exemplo, em código fonte, que respeita a atual especificação de linguagem Java, compatível com a atual versão Java, J2SE 6.

```
package examples.jigsaw;

public class HelloWorld {

  static public void main(String args[]) {
    System.out.print("Hello world!\n");

  }
}
```

Tabela 4.3.1: Exemplo HelloWorld em código fonte Java

Esta classe – HelloWorld – pertencente ao package "examples.jigsaw", compila e executa sem qualquer alteração, na plataforma J2SE 6.

Se pretendermos modularizar esta classe, criando por exemplo um módulo com nome "HelloWorld" constituído pela classe "examples.jigsaw.HelloWorld", definimos um ficheiro de módulo – "module-info.java" – tal como indicado a seguir.

```
module HelloWorld@1 provides HelloWorldGlobal@1 {
  requires local jdk.base @ 7-ea;
  permits HelloWorldOtherModule1, HelloWorldOtherModule2;
  class examples.jigsaw.HelloWorld;
}
```

Tabela 4.3.2: Definição do módulo HelloWorld

Esta unidade de compilação define que o módulo "HelloWorld" depende do módulo "jdk.base", versão "7-ea" e que ambos o módulos devem ser carregados utilizando o mesmo ClassLoader, devido ao atributo "local" utilizado. Adicionalmente, o módulo HelloWorld apenas pode ser declarado como dependência pelos módulos "HelloWorldOtherModule1" e "HelloWorldOtherModule2".

No processo de compilação, as classes em código fonte e o módulo são compilados. A partir desse momento, todas as classes são marcadas como pertencendo ao módulo definido.

```
javac module-info.java examples/jigsaw/HelloWorld.java
```

Tabela 4.3.3: Compilação do módulo HelloWorld

O módulo é instalado depois num repositório, utilizando por exemplo a nova ferramenta "jmod" que vai fazer parte das ferramentas de desenvolvimento do Jigsaw e, para executar o módulo basta evocar o comando:

```
java -m HelloWorld
```

Tabela 4.3.4: Execução do módulo HelloWorld

Conforme se demonstra, o código fonte continua a ser escrito da mesma forma, de forma completamente abstraída de módulos.

É no ato da compilação que definimos, através da compilação conjunta do módulo e do resto do código fonte, as classes e a informação do módulo.

Desta forma, a escrita de aplicações modulares, com recurso à metodologia Jigsaw, não necessita de alterações ao código fonte e necessita apenas de:

- Definição dos módulos através dos ficheiros "module-info.java"

- Compilação de todo o código incluindo a definição do módulo

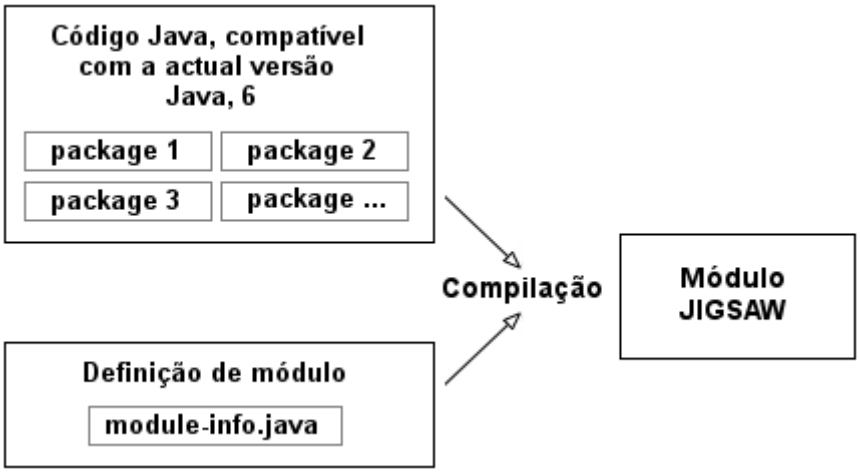


Figura 4.3.1: Processo de implementação em Jigsaw

- Instalação e posterior execução do módulo na biblioteca

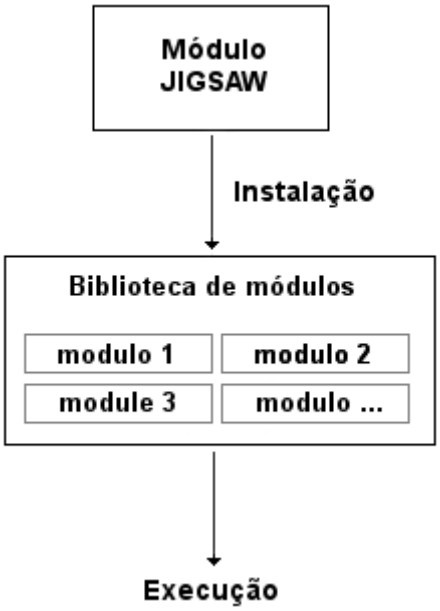


Figura 4.3.2: Instalação e execução de módulo

Desta forma, valida-se que é possível utilizar a metodologia modular Jigsaw para escrita de aplicações modulares Java.

4.3.1 Limitações

Existem no entanto limitações que, não impedem a utilização da metodologia mas dificultam a sua utilização em alguns cenários.

Podem identificar-se os seguintes aspetos que não estão previstos no Jigsaw e que, em certos cenários de desenvolvimento, podem ser restritivos, se se pretender utilizar a metodologia modular.

O primeiro, identifica-se pelo facto que, no processo de desenvolvimento e instalação de módulos, existe um processo que consiste em instalar os módulos numa biblioteca. O projeto Jigsaw define apenas uma biblioteca de módulos, a da própria plataforma mas não define em que biblioteca os módulos aplicacionais externos à plataforma devem ser instalados. A própria versão atual Jigsaw utiliza uma outra biblioteca para instalação dos módulos adicionais mas sem qualquer processo que defina formalmente onde deve residir esta biblioteca.

O segundo consiste em que não está definida a forma como os ficheiros que constituem os módulos serão arquivados para efeitos de distribuição.

O terceiro identifica-se pela inexistência de mecanismos que permitam aos próprios módulos ter noção de eventos importantes na sua existência: instalação; ativação; desativação; desinstalação – a instalação nas bibliotecas, a ativação, desativação dos módulos está definida para ser completamente transparente, o que se torna uma mais valia por simplificar todo o processo. No entanto, em sistemas de software, torna-se necessário muitas vezes que os componentes constituintes possam perceber as várias fases do ciclo de vida e tomar ações, nomeadamente preparar determinadas configurações iniciais após instalação ou eliminar essas configurações aquando da desinstalação.

Por último, é importante indicar que a estrutura de diretorias onde reside a plataforma Java sofre alterações e deixam de existir ficheiros e diretorias que tipicamente existiam até agora. Esta diretoria corresponde, na especificação de máquina virtual Java, à propriedade de plataforma Java “jdk.home”. Uma vez que a plataforma se encontra modularizada, as classes e bibliotecas Java deixam de estar localizadas nos ficheiros e diretorias tradicionalmente existentes e passam a estar arquivados em bibliotecas de módulos. Esta alteração é

importante uma vez que existem muitas aplicações que validam a existência de ficheiros especiais, como é o caso do ficheiro "rt.jar" ou diretorias especiais, como é o caso da diretoria "lib", situação que deixa de ser possível.

Estas limitações, identificadas, não são impeditivas de adotar, desde já, a metodologia para desenvolvimento aplicacional modular embora possam representar a necessidade de algumas adaptações em determinados cenários de desenvolvimento, nomeadamente e principalmente em cenários de modularização de código já existente.

4.4 Integração e *deploy* em servidores de aplicações

Atualmente, em sistemas aplicacionais com elevado número de sub-sistemas, como é o caso dos servidores aplicacionais, já são geralmente adotadas estratégias de modularização, algumas com base nos processos descritos no Capítulo 2.

Pretende-se demonstrar a viabilidade de se utilizar a metodologia modular Jigsaw para implementar servidores aplicacionais.

Uma vez que cada servidor aplicacional adota geralmente metodologias de implementação distintas, o foco deste estudo terá como base o servidor de aplicações JBoss AS, versão 5. Este servidor implementa a especificação Java Enterprise Edition 5 (JEE 5) e é um dos servidores aplicacionais mais utilizado no mercado empresarial. O foco neste servidor não invalida o estudo uma vez que, embora sejam adotadas metodologias distintas pelas empresas que desenvolvem os vários servidores, os conceitos principais estão sempre presentes pelo que o estudo aplicado num servidor em concreto é passível de se aplicar noutros servidores.

4.4.1 Modelo aplicado aos servidores aplicacionais

Um servidor aplicacional caracteriza-se, tipicamente, por ser constituído por vários níveis ou camadas funcionais.

Estas camadas fornecem serviços que suportam o desenvolvimento de aplicações que permitem resolver os problemas que se colocam aos sistemas de informação.

O JBoss AS implementa estes serviços através de uma arquitetura designada por JBoss Microcontainer[25] [Johnson 09]. Por razões históricas, o JBoss implementa adicionalmente uma camada básica de serviços, designada por *micro-kernel*[26] JMX[27] [Johnson 09]. Estas duas funcionalidades asseguram a existência de uma infraestrutura básica de gestão de componentes e proporcionam as seguintes funções básicas: i) gestão de instanciação de objetos; ii) gestão de dependências entre componentes e objetos; iii) gestão de propriedades dos objetos e componentes; iv) especificação da forma de definir o modelo do ponto de vista de configuração [Johnson 09].

Com estas funcionalidades básicas é possível o desenvolvimento de componentes que asseguram os serviços necessários à implementação do servidor, como sejam serviços de passagem de mensagens, os serviços de implementação da plataforma JEE, os serviços de carregamento e descarregamento de aplicações.

Estas funcionalidades asseguram o modelo de plugins/módulos do JBoss AS.

Podemos, identificar 2 grandes grupos de funcionalidades.

O primeiro grupo, o grupo de funções nucleares, são as funções nucleares do servidor, que incluem a gestão do processo de arranque e paragem da aplicação, que podem incluir todas as funcionalidades que caracterizam o servidor e fazem a gestão de configurações e definição de todo o servidor, funcionalidades de gestão de *plugins*, módulos e componentes. Essas funcionalidades podem incluir funções de carregamento e descarregamento de componentes. No JBoss, este

25 Microcontainer poderá traduzir-se em português por micro-contentor. O termo Microcontainer será utilizado na dissertação por refletir de forma mais exata a especificação JBoss AS.

26 Micro-kernel pode traduzir-se em português por micro-núcleo. O termo micro-kernel é utilizado na documentação original do JBoss AS e será utilizada ao longo da dissertação por melhor traduzir a versão original da documentação do JBoss AS.

27 JMX – Java Management Extensions, em português, extensões de gestão Java. Trata-se de uma especificação Java, aprovada no JCP com o JSR 3.

grupo é constituído pelo JBoss Microcontainer e pelo micro-kernel JMX. O JBoss é um dos servidores que implementa e faz utilização de funcionalidades de gestão dinâmica de componentes, para permitir uma gestão apurada de todo o ambiente aplicacional. A gestão dinâmica de componentes permite ao JBoss instalar, remover, carregar e descarregar componentes sempre que seja necessário, mediante condições específicas ao longo da execução do servidor.

O segundo grupo, constituído pelos componentes, *plugins* e módulos, define as funcionalidades adicionais necessárias aos servidores – a camada de serviços. Estes componentes permitem que os servidores suportem várias especificações e normas, permitindo-se adaptar a situações concretas. No JBoss estes componentes asseguram os serviços de registo de eventos (logging), gestão de mensagens, camadas de serviços web, a implementação das APIs JEE, entre outros, de uma lista extensa [Marchioni 09].

Desta forma, o servidor JBoss pode ser visto como sendo constituído por "***m***" grupos de funcionalidades nucleares "**F**", e por "***o***" componentes "**C**", que implementam as camadas de serviços:

$$\sum_{m} F + \sum_{o} C$$

O JBoss AS, tal como outros servidores, adota assim de raiz um conceito estruturado e modular de gestão e suporte de componentes. No entanto e como já foi referido nos capítulos anteriores, os modelos são avançados mas incompatíveis entre si.

Para demonstrar de que forma podemos aplicar a metodologia Jigsaw, é realizada uma abordagem por equiparação.

Pretendemos demonstrar que, tendo como base a arquitetura de um servidor desenvolvido em Java, o JBoss AS, é possível implementar cada uma das camadas com metodologia modular Jigsaw, sem se perder qualquer funcionalidade.

A abordagem por equiparação pretende demonstrar que, partindo de um servidor já implementado, é possível, através de processos de adaptação e

transformação, transforma-lo num modelo modular.

No modelo tradicional, independentemente da abordagem de desenvolvimento adotada, o objetivo é implementar um conjunto de especificações que permitam disponibilizar a plataforma de serviços para implementação de aplicações.

Aplicando regras podemos definir o processo de transformação.

As funções nucleares, compreendem algumas funcionalidades básicas, nomeadamente de gestão de arranque e paragem do servidor, e outras funcionalidades de gestão de componentes. Podemos implementar estas funções num ou mais módulos, sendo esses módulos os módulos base do servidor. Todas as funcionalidades de gestão de plugins e componentes devem ser omitidas nesta transformação. Toda a gestão de dependências e carregamento de módulos passa a ser um problema de inteira responsável da máquina virtual Java. Num modelo extremo, podemos implementar todas as funcionalidades num único módulo. No entanto, podemos eliminar toda a lógica de gestão de componentes uma vez que as suas funcionalidades se sobrepõem às do Jigsaw. Sendo assim, tendo como base o conjunto de "***m***" funcionalidades "**F**", podemos, por modularização, definir o conjunto "***n***" módulos de funcionalidades "**MF**".

$$\sum_{m} F \rightarrow \sum_{n} MF$$

A forma como podemos definir módulos a partir de código Java foi demonstrada na secção 4.3.

Todos os componentes e *plugins* do 2º grupo de funcionalidades, são identificados e transformados em módulos. Este processo é mais simples que o anterior uma vez que se pretende uma relação direta entre componentes e módulos Jigsaw. Estes componentes têm apenas que sofrer pequenas alterações, que permitam eliminar quaisquer dependências ao modelo de gestão de módulos proprietário. Todo o outro código mantém-se. O processo consiste em, para cada componente, definir o ficheiro de definição do módulo "module-info.java" e recompilar o componente, com o ficheiro "module-info.java", por forma a obter o módulo Jigsaw compilado. Sendo assim, tendo como base "***o***"

componentes "**C**", transforma-se esses componentes em "***o***" módulos Jigsaw "**M**".

$$\sum_{o} C \quad \rightarrow \quad \sum_{o} M$$

O resultado final será um servidor aplicacional constituído por módulos,

$$\sum_{n} MF + \sum_{o} M$$

que definem o servidor.

A maior complexidade no processo de transformação reside na limpeza de todo o código relacionado com o sistema de gestão de componentes e plugins.

Das funcionalidades básicas disponibilizadas pelo JBoss AS – i) gestão de instanciação de objetos; ii) gestão de dependências entre componentes e objetos; iii) gestão de propriedades dos objetos e componentes; iv) especificação da forma de definir o modelo do ponto de vista de configuração – apenas a funcionalidade relacionada com a gestão de propriedades dos objetos e componentes deverá persistir. Todas as outras, sobrepõem-se de alguma forma ao Jigsaw e devem ser eliminadas. Isto significa um esforço de análise e alteração do código elevado, no entanto possível. Numa situação normal, o servidor aplicacional será desenvolvido pensando já segundo a metodologia Jigsaw, tornando a sua arquitetura mais clara e simples.

É possível criar com grande simplicidade módulos, a partir de código fonte existente. O facto de não ser necessário ao próprio servidor gerir funcionalidades de plugins e componentes simplifica ainda mais o código. As dependências e carregamento automático de módulos passa a ser gerido automaticamente pela maquina virtual Java.

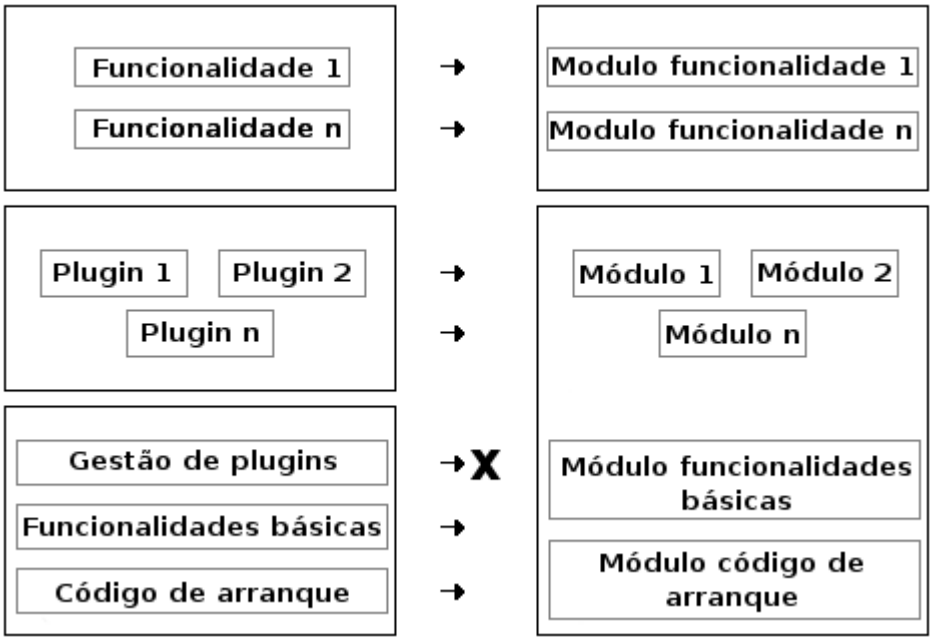


Figura 4.4.1: Modelo de aplicabilidade de metodologia Jigsaw no desenvolvimento e integração de um servidor

Valida-se que é possível desenvolver, do ponto de vista programático, servidores aplicacionais recorrendo à metodologia Jigsaw.

No entanto, e utilizando o JBoss AS como modelo no processo de transformação, uma das funcionalidades que o JBoss AS assegura na gestão de módulos é a de gestão dinâmica de componentes. São funcionalidades que assumem especial importância nos servidores e em aplicações de elevada complexidade dos serviços prestados uma vez que são as que permitem que as próprias aplicações, durante a execução, se adaptem dinamicamente.

Analisando com mais detalhe estas funcionalidades, podemos caracterizar essas funcionalidades por: i) gestão dinâmica do ciclo de existência dos componentes (instalação, desinstalação, carregamento e descarregamento, incluindo durante a execução do servidor); ii) gestão de dependências de componentes.

O modelo Jigsaw, no que refere à gestão de módulos, implementa todas as características que o JBoss AS implementa, mas com abordagens distintas. Uma das abordagens que o Jigsaw adota é que não permite uma gestão dinâmica de

módulos, não permitindo por exemplo a instalação e carregamento de módulos durante a execução de uma aplicação. Adicionalmente, o Jigsaw não exporta uma qualquer API pública que permita gerir o processo de gestão de módulos. De forma complementar, torna-se mais evidente um problema identificado na secção 4.3, que consiste na incapacidade dos módulos serem notificados de eventos importantes relacionados com a gestão de módulos (instalação, desinstalação, carregamento, descarregamento). Estas funcionalidades são muito importantes no JBoss AS e em outros servidores para conseguir implementar a gestão dinâmica de componentes, permitindo a estes realizar operações importantes em cada uma das fases da sua existência, nomeadamente guardando dados importantes de estado para mais tarde serem recuperados.

Numa arquitetura tradicional, o servidor tem funcionalidades que permitem instalar, ativar, desativar e desinstalar componentes e plugins de forma programática, permitindo assim que se auto-modele de forma dinâmica durante a execução. Adicionalmente complementa todos estes processos com eventos que permitem notificar partes da aplicação acerca destes acontecimentos.

Das atuais especificações Jigsaw, nenhuma responde a estas necessidades. Uma análise pormenorizada do código fonte do projeto revela que algumas das funcionalidades estão presentes mas não exportadas como APIs para as camadas aplicacionais. Desta forma, não podem ser utilizadas pelas aplicações de forma formal e segura.

As funcionalidades de instalação e desinstalação apenas podem ser realizadas com a aplicação inativa e a ativação e desativação dos módulos é realizada de forma estática pela máquina virtual.

A ativação e desativação de módulos não é possível durante a execução das aplicações. Este processo é assegurado automaticamente pela máquina virtual Java. Sem estas funcionalidades, não é possível a implementação de servidores segundo os requisitos definidos sem que para tal se percam funcionalidades criticas para o funcionamento dos próprios servidores.

4.4.2 Conclusões da metodologia em servidores aplicacionais

Do estudo de integração de módulos em servidores, percebemos as mesmas limitações já encontradas na secção 4.3.

A primeira consiste na inexistência de definição das bibliotecas onde devem ser instalados os módulos aplicacionais. Esta aspeto foi identificado na secção 4.3 e mantêm-se o problema com os servidores de aplicações. Não é um problema crítico e impeditivo mas tem que ser dada a devida importância para evitar dispersão nas implementações.

A segunda resulta de que não está definida a forma como os ficheiros que constituem os módulos serão arquivados para efeitos de distribuição. Esta situação foi identificada na secção 4.3 e torna-se mais evidente a sua ausência nos servidores aplicacionais, que trabalham com muitos componentes.

A terceira consiste na inexistência de mecanismos que permitam aos módulos aperceber-se de eventos importantes no processo de gestão de módulos: instalação; ativação; desativação; desinstalação – a instalação nas bibliotecas, a ativação, desativação dos módulos está definida para ser completamente transparente, o que se torna uma mais valia. No entanto, resultante da necessidade de grande dinamismo deste tipo de aplicações, existe a necessidade dos módulos constituintes da aplicação de serem notificados de acontecimentos importantes no processo de execução, que permita realizar operações vitais, tais como guardar dados de estado, carregar dados de estado ou definir as configurações iniciais após instalação. Será muito difícil implementar servidores sem estas facilidades.

Adicionalmente e ainda mais importante, não é possível realizar as tarefas básicas de gestão de módulos (instalação e ativação) durante a execução das aplicações.

Conclui-se que não será possível utilizar a metodologia sem abdicar de algumas funcionalidades consideradas criticas no desenvolvimento de servidores aplicacionais pelo que nesta fase, não é viável aplicar a metodologia no desenvolvimento de servidores.

Estas melhorias não estão implementadas nem estão previstas pelo que, conclui-se, a aplicabilidade da metodologia no desenvolvimento de servidores aplicacionais e aplicações com arquiteturas complexas (elevado número de módulos, arquitetura reflexiva) fica limitada e pode-se considerar, inviabilizada de ser aplicável em sistemas comerciais ou críticos.

Capítulo 5

Alterações propostas

5.1 Introdução

No Capítulo 4 foram identificadas limitações várias ao modelo de desenvolvimento Jigsaw.

Com base nas limitações identificadas, o Capítulo 5 apresenta um conjunto de melhorias que procuram fortalecer a metodologia modular Jigsaw para desenvolvimento de aplicações Java, definindo um conjunto de especificações e regras que tornam todo o processo de desenvolvimento modular mais robusto.

É feito um estudo e são propostas alterações e melhorias que têm como objetivo definir: i) a forma como as bibliotecas de módulos devem estar organizados num sistema; ii) as regras que se devem aplicar na gestão dessas bibliotecas; iii) identificar um modelo que permita implementar um mecanismos básico de notificações de eventos relacionados com a máquina de estados modular; iv) implementar um conjunto de funcionalidades que permita gerir dinamicamente e programaticamente os módulos e processos associados.

5.2 Bibliotecas

A proposta de melhoria respeitante às bibliotecas de módulos assenta na definição de um conjunto mínimo de bibliotecas, que servem de base à execução das aplicações. Este elemento permite definir um conjunto de regras de gestão

que permite facilitar todo o modelo de desenvolvimento modular.

A motivação pela definição exata do conjunto de bibliotecas que devem num ambiente resulta da necessidade de definir perfeitamente o âmbito de gestão de repositórios e bibliotecas, evitando dispersão de metodologias de gestão de bibliotecas. Adicionalmente, representa um mapeamento com os atuais processos de gestão de software dos sistemas operativos, nomeadamente Windows, Linux e Mac.

O modelo define que, num sistema aplicacional, têm que existir no mínimo duas bibliotecas distintas:

- **Biblioteca de plataforma**
- **Biblioteca de sistema operativo**

Adicionalmente, podem existir mais duas bibliotecas, opcionais:

- **Biblioteca de aplicação**
- **Biblioteca de utilizador**

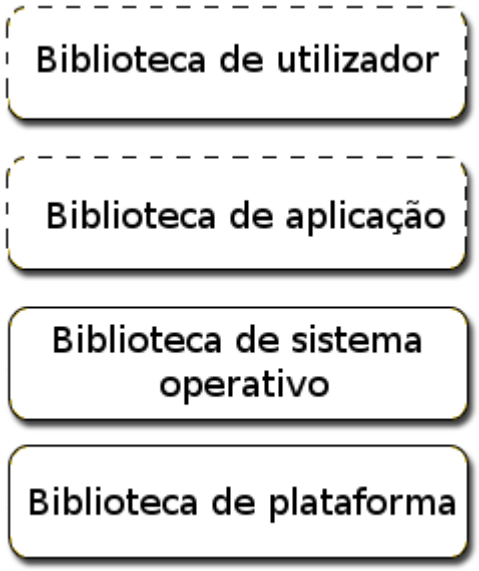


Figura 5.2.1: Hierarquia de bibliotecas

As bibliotecas relacionam-se de forma hierárquica respeitando as regras definidas a seguir.

5.2.1 Biblioteca de plataforma

Esta biblioteca contêm todos os módulos da plataforma Java. É o resultado da instalação da plataforma Java no sistema operativo e proporciona todos os módulos base da plataforma Java, que disponibilizam as APIs base tais como *java.lang.** e *java.io.**.

Por definição, não pode conter nenhum módulo adicional para além dos módulos distribuídos com a distribuição Java e que fazem parte da plataforma base.

Esta biblioteca é a única que atualmente está definida na atual implementação Jigsaw. No ambiente Jigsaw é designado por **biblioteca de sistema**.

5.2.2 Biblioteca de sistema operativo

Esta biblioteca contêm todos os módulos das aplicações Java instaladas no sistema operativo que não fazem parte da plataforma Java.

Define-se os seguintes aspetos:

- O biblioteca deve estar localizado numa área que possa estar acessível por todas as aplicações e todos os utilizadores do sistema operativo.

- Nesta biblioteca, estão instalados todos os módulos que representam as várias aplicações Java instaladas, incluindo as aplicações que exportam funcionalidades para o utilizador final ou para o sistema operativo assim como sistemas software com objetivo final de facultar funcionalidades a outras aplicações, nomeadamente APIs de acesso a bases de dados.

- Sempre que possível, a gestão deste repositório tem que estar apenas reservada a um grupo de utilizadores restrito com permissões para tal,

sejam administradores do sistema operativo sejam utilizadores com permissões adequadas ao efeito, recorrendo a gestão das permissões dos ficheiros em disco ou a qualquer outra política de gestão de permissões do sistema operativo.

- A biblioteca de sistema operativo tem que ser criada e definida tendo como dependente da biblioteca de plataforma.

5.2.3 Biblioteca de aplicação

Esta biblioteca contêm módulos que, dada a especificidade, por exemplo a da aplicação, são instalados numa biblioteca completamente distinta.

Com este método é possível instalar aplicações na biblioteca de sistema operativo e adicionalmente, instalar módulos específicos e restritos numa biblioteca adicional.

Por definição, quando uma aplicação é instalada na biblioteca de aplicação, tem que definir como biblioteca dependente a biblioteca do sistema operativo.

5.2.4 Biblioteca de utilizador

Esta biblioteca contêm módulos que, ficam apenas disponíveis em bibliotecas alocadas a cada um das contas dos utilizadores.

Com este método é possível instalar aplicações na biblioteca de utilizador, de forma autónoma e independente, por cada utilizador, sem necessidade de permissões adicionais.

Por definição, deve ser possível a instalação de aplicações na biblioteca de utilizador e, quando uma aplicação é instalada na biblioteca de utilizador, tem que definir como biblioteca dependente a biblioteca de sistema operativo.

5.2.5 Regras semânticas

Aplicam-se as seguintes definições semânticas, no processo de desenvolvimento

de software e na organização e gestão das bibliotecas de módulos apresentadas:

- Existe um conjunto de bibliotecas, bem definido, que tem que constar em todos os sistemas e cuja existência todas as aplicações têm que respeitar: biblioteca de plataforma e biblioteca de sistema operativo.

- Para além destas bibliotecas, as aplicações podem utilizar bibliotecas adicionais, restritas ao âmbito delas próprias: biblioteca de aplicação.

- Nos sistemas operativos com capacidades multi-utilizador, cada utilizador pode ter uma biblioteca própria, relacionada apenas com a sua conta de utilizador: biblioteca de utilizador.

5.3 Sistema de notificações

Para além da sistematização de bibliotecas proposta, é proposto também melhorias que permitem implementar funcionalidades de notificações relacionadas com eventos importantes do processo de módulos.

A metodologia divide-se em 3 alterações fundamentais:

- Estender a definição de módulo para identificar os objetos disponíveis no módulo para receber eventos.

- Definir o tipo de eventos existentes

- Alterar a máquina virtual para que esta passe a gerar os eventos necessários nas várias fases do processo

5.3.1 Extensão da definição de módulo para suporte de observador de eventos

Propõe-se a adição de mais um atributo, para suportar a possibilidade de, no ato da definição do módulo, indicar o conjunto de objetos que pretendem ser notificados de eventos relacionados com o processo de gestão de módulos.

O atributo proposto é "**listener**" e a sintaxe é: ***listener class1[, classn];***

Por exemplo, tendo como base o exemplo de módulo HelloWorld, assumindo que pretendemos que a classe " examples.jigsaw.HelloWorld" seja notificada de eventos, o process será definido como se segue:

```
module HelloWorld@1 provides HelloWorldGlobal@1 {
  requires local jdk.base @ 7-ea, jdk.swing @ 7-ea;
  permits HelloWorldOtherModule1, HelloWorldOtherModule2;
  class examples.jigsaw.HelloWorld;
  listener examples.jigsaw.HelloWorld;
}
```

Tabela 5.3.1: Modelo proposto para definição de observadores de eventos

Este modelo é simples e exige poucas alterações nos analisadores gramaticais. O que se pretende é que, por exemplo neste caso, o objeto examples.jigsaw.HelloWorld tenha um construtor sem parâmetros que permita ao sistema instanciar um objeto diretamente a partir da classe examples.jigsaw.HelloWorld e, esta classe, ao implementará um *interface* Java com os métodos específicos, ficar apta a ser notificada de eventos importantes relacionados com o processo de modularidade.

A seguir apresenta-se o interface proposto.

5.3.2 *Definição de eventos possíveis*

São identificados o conjunto de eventos considerados importantes:

Evento	Explicação do evento
moduleInstalled(Library l, Module m, boolean before)	Notificação de módulo instalado
moduleUninstalled(Library l, Module m, boolean before)	Notificação de módulo desinstalado
moduleActivated(Library l, Module m, boolean before)	Notificação de módulo ativado no ambiente Java
moduleDeactivated(Library l, Module m, boolean before)	Notificação de módulo desativado no ambiente Java

Tabela 5.3.2: Eventos para notificação

Estes eventos devem ser implementados num tipo "interface" e instanciados pelas classes que se querem tornar observadoras destes eventos.

Todos os eventos têm as versões "before" e não "before", indicando precisamente duas fases: antes do evento se concretizar e depois do evento se concretizar.

Por exemplo, o evento "moduleInstalled", para um módulo que vai ser instalado numa biblioteca, é gerado 2 vezes para esse mesmo módulo: i) a 1ª, antes do módulo ser instalado, com o parâmetro "before" com valor verdadeiro; ii) a 2ª, depois do módulo ser instalado, com o parâmetro "before" com valor falso.

Com este interface simples, será possível a cada módulo acompanhar o processo de gestão modular e agir em conformidade com os interesses próprios.

Para concretizar este interface, torna-se necessário alterar a máquina virtual, compilador e ferramentas de suporte, para que, em cada fase de gestão de módulos identificadas no ponto anterior, notifiquem todas os observadores de eventos registados. As ferramentas de gestão de módulos têm que analisar a informação de módulo e gerar os eventos correspondentes sempre que seja necessário: i) moduleInstalled; ii) moduleUninstalled.

Uma análise ao código fonte atual demonstra que, a tarefa não implicará um

esforço demasiado grande, sendo perfeitamente viável de implementar uma vez que todas as fases identificadas se podem mapear para dentro do código do compilador e máquina virtual, já que, de certa forma, estes já têm que ter a noção destas fases internamente.

5.4 Gestão de carregamento e descarregamento de módulos

Este ponto de melhoria foca a atenção em algo que hoje em dia é possível de utilizar em Java, que é o carregamento e descarregamento dinâmico e programático de classes. Os servidores aplicacionais fazem-no com grande frequência, nomeadamente ao nível formal dos componentes ou plugins.

O Jigsaw define que os módulos são carregados de forma estática pela máquina virtual e que esse processo não é controlável programaticamente.

O que se apresenta é um conjunto simples de funcionalidades que deverá permitir, de forma programática, controlar a instalação, desinstalação, carregamento e descarregamento de módulos. Estas funcionalidades são importantes na implementação de sistemas dinâmicos, como é tipicamente o caso de servidores aplicacionais.

Propõe-se a adição de 4 funções à API de gestão de módulos:

Função	Explicação da função
loadModule	Força o carregamento explicito de um módulo
unloadModule	Liberta o módulo de memória
installModule	Instala um módulo
uninstallModule	Desinstala um módulo

Tabela 5.4.1: Funções de controlo programático do processo modular

Após análise do código fonte Jigsaw, verifica-se que a implementação destas

funcionalidades será acima de tudo um processo de externalização de funcionalidades já presentes no código. Estas funcionalidades existem mas estão definidas no âmbito de APIs internas que não fazem parte das APIs a publicar publicamente.

5.5 Conclusões

Com a metodologia Jigsaw percebe-se a simplicidade de escrita de módulos. O código existente não tem que ser alterado. Não necessitamos programar com alguma API especial. Não necessitamos de manter uma infraestrutura de gestão de módulos distinta da máquina virtual. E todo o processo está pronto a ser adaptado desde o momento inicial. Os únicos requisitos: modelar os módulos, definir as dependências e alterar os processos de compilação para ter em consideração a nova metodologia.

Verifica-se que a metodologia pode ser aplicada sem constrangimentos para a desenvolvimento aplicacional Java, nomeadamente o de sistemas complexos, como é o caso de servidores aplicacionais Java.

No entanto, uma análise mais pormenorizada, tendo como base estudos realizados em plataformas servidores atualmente implementados, revelam que existem limitações que podem dificultar a aplicabilidade da metodologia num projeto de desenvolvimento, onde existem requisitos de carregar e descarregar módulos dinamicamente e com elevado número de módulos e interdependências entre si, como é o caso dos servidores aplicacionais. É essa a motivação que leva a apresentar as melhorias proposta, que, espera-se, permitam resolver as limitações identificadas.

Capítulo 6

Software de modelação de módulos Jigsaw

No âmbito desta dissertação, um dos objetivos é desenvolver uma aplicação informática (prova de conceito) que, mediante circunstâncias perfeitamente definidas, permita facilitar a modularização de projetos segundo a metodologia Jigsaw.

6.1 Introdução

Com esse objetivo, foi desenvolvida uma aplicação informática, designada de Gestor e Modelador de Módulos Jigsaw (GMMJ), escrita na linguagem Java e que tem como especificações base permitir a modularização de projetos escritos na linguagem Java, segundo o modelo Jigsaw, de forma simples e o mais automatizada possível.

Durante a fase de planeamento, foi necessário decidir se a aplicação ia ser desenvolvida utilizando a metodologia de desenvolvimento de módulos Jigsaw ou se seria desenvolvida segundo uma metodologia compatível com o J2SE 6. Foram realizados vários testes e tornou-se evidente que a atual versão executável da plataforma Jigsaw não apresenta a estabilidade, funcionalidades e compatibilidade com ferramentas de desenvolvimento necessárias pelo que a aplicação foi desenvolvida em Java, na plataforma J2SE 6, segundo as metodologias atuais Java.

A decisão teve como base de fundamento a dificuldade sentida resultante da tentativa de desenvolver a aplicação tendo como base a plataforma Java, versão Jigsaw: Embora todas as APIs base da plataforma Java estejam presentes através dos packages standard, nomeadamente *java.lang.**, *java.io.**, o desenvolvimento de uma aplicação, por simples que possa ser, pode implicar a utilização de funcionalidades externas à plataforma, desenvolvidas por terceiros, quer seja para melhorar o aspeto gráfico, quer seja para aceder a bases de dados ou por qualquer outro motivo. Embora esses packages estejam presentes através de ficheiros JAR, a sua utilização não é simples e atualmente não é sequer possível uma vez que não estão implementadas em módulos e seria necessário recompilar todas as bibliotecas necessárias, criando versões de módulos, situação que pode traduzir-se numa tarefa complexa e com dependências de terceiros – os implementadores dessas bibliotecas – eventualmente com custos associados. Uma alternativa será o desenvolvimento e execução das aplicações utilizando o modo híbrido descrito na subsecção 3.4. Este modo não está no entanto implementado pelo que a sua utilização não é possível.

Durante o desenvolvimento da aplicação e para demonstrar as funcionalidades, a própria aplicação em código fonte é modularizada segundo o modelo Jigsaw.

6.2 Estrutura da programação

O programa divide-se em duas camadas: i) camada de lógica funcional ii) camada de apresentação.

6.2.1 Camada de lógica funcional

Esta camada, que implementa abstração completa de interface de utilizador, inclui todos os packages e classes base necessários à implementação das funcionalidades lógicas do programa: modularização e gestão de bibliotecas.

Esta camada é constituída por 2 packages:

- **org.uminho.di.mi2010.jigsaw.logic**

Classes	Âmbito
JMSDController	Controlador de todas as funcionalidades de modularização
JMSDControllerException	Classe indicadora de erro geral no âmbito do controlador
JMSDControllerResourceAlreadyExistsException	Classe indicadora de erro relacionado com a tentativa de executar operações de inserção de recursos já previamente inseridos, no âmbito do controlador
JMSDControllerResourceNotFoundException	Classe indicadora de erro relacionado com a tentativa de referência a um recurso inexistente, no âmbito do controlador

Tabela 6.2.1: Package de lógica funcional do GMMJ

- **org.uminho.di.mi2010.jigsaw.logic.objects**

Classe	Âmbito
Library	Representação de uma biblioteca Jigsaw
Module	Representação de um módulo Jigsaw
ModuleDependency	Representação de uma dependência entre módulos
ModuleProvide	Representação de uma relação de disponibilização (*provide*) de um módulo
Package	Representação de um package

Tabela 6.2.2: Package de objetos de suporte à lógica funcional do GMMJ

Esta camada assegura a gestão de todas as funcionalidades de importação de bibliotecas, de importação de packages, de manutenção de projetos e de resolução automática de dependências entre módulos assim como o próprio processo de geração de módulos (ficheiros module-info.java).

Para implementar processo de resolução automática de dependências, o controlador enriquece a base de dados interna da aplicação com todas as classes importadas pelos packages do projeto a modularizar. Desta forma, consegue calcular, com base nos packages que constituem o módulo e tendo conhecimento de todos os packages que constituem os outros módulos, todas as dependências entre módulos.

Durante a importação dos packages que fazem parte do projeto, o respetivo código fonte é reconhecido, utilizando uma biblioteca externa e, são determinados os tipos (classes e interfaces) utilizados pelos packages e que são externos a cada um dos packages.

No âmbito de cada projeto, a aplicação tem sempre conhecimento de todos os tipos necessários a cada package e de todas as dependências entre módulos, sejam eles módulos do próprio projeto ou das bibliotecas importadas.

Todas as operações são realizadas tendo ponto único o controlador - JMSDController.

6.2.2 Camada de apresentação

Esta área representa todos os packages e classes necessários à criação da lógica de apresentação, que permite disponibilizar aos utilizadores todas as funcionalidades necessárias.

O modelo de apresentação é baseado no conjunto de APIs Swing[28], permitindo a utilização de APIs já presentes na plataforma Java.

A aplicação é constituída uma uma janela principal, onde podem ser abertas uma ou mais janelas interiores e caixas de introdução de dados, representando as várias funcionalidades.

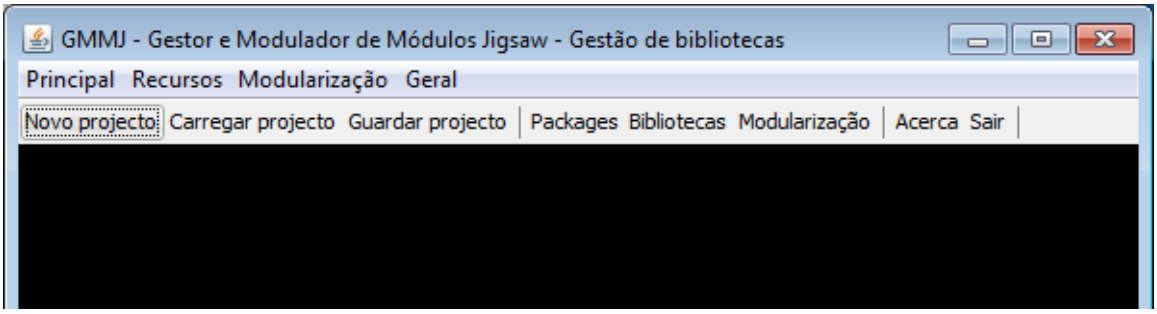


Figura 6.2.1: Opções da janela principal da aplicação GMMJ

As funcionalidades agrupam-se em 4 grupos apresentados a seguir.

28 API Swing é um conjunto de APIs presente na plataforma Java SE 6 e que permite desenvolver interfaces de utilizador gráficas.

6.2.2.1 Gestão dos projetos de modularização

Este grupo de funcionalidades permite a criação, abertura e salvaguarda de projetos. Permite também a ligação a bibliotecas de módulos. Estes módulos são necessários para enriquecer o processo de modularização.

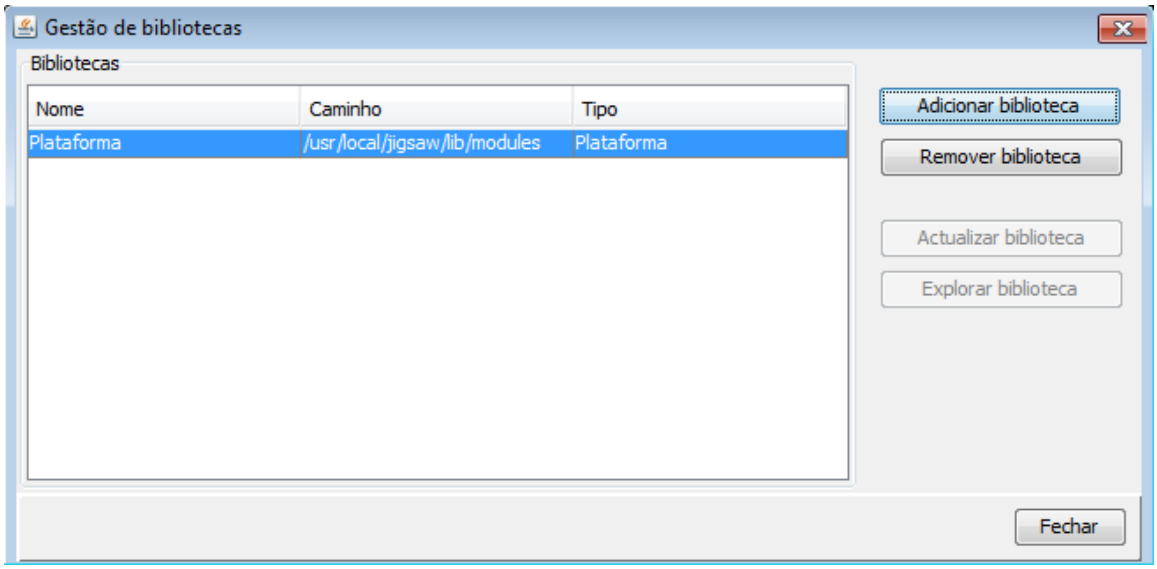


Figura 6.2.2: Janela de gestão de bibliotecas

É possível importar bibliotecas para o projeto, de acordo com o modelo apresentado na tese: bibliotecas de plataforma; bibliotecas de sistema operativo; bibliotecas de utilizador e bibliotecas de aplicação.

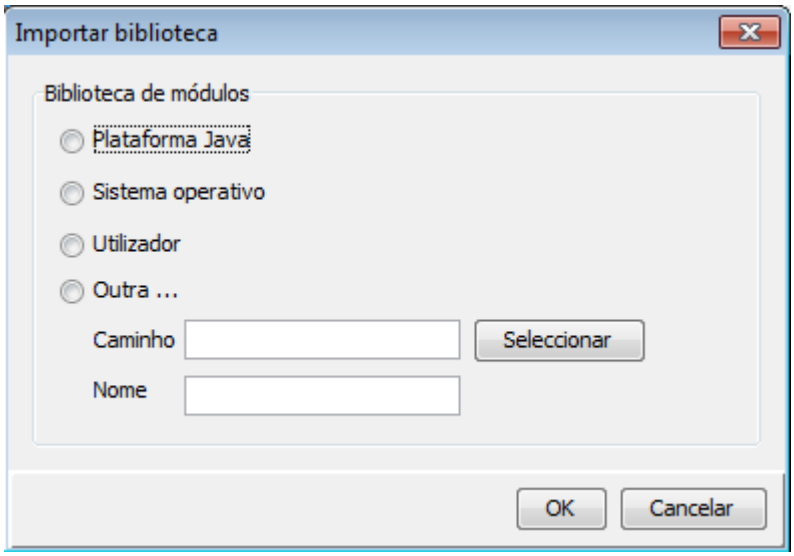


Figura 6.2.3: Caixa de dialogo para importação de biblioteca

É também nesta secção que são importadas as classes de código fonte que constituem os projetos Java a modularizar, tal como se indica na figura seguinte.

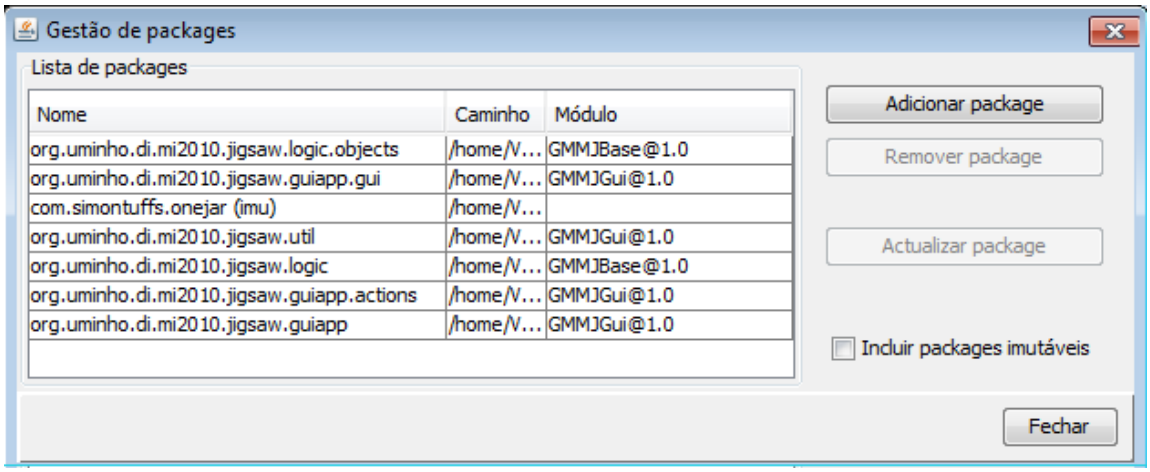


Figura 6.2.4: Janela de gestão de packages a modularizar

Durante o processo de importação, são identificados todos os tipos importados por cada classe e interface pertencentes aos packages, para mais tarde essa informação ser utilizada para calculo automático de dependências entre módulos.

6.2.2.2 Gestão do processo de modularização

Este grupo de funcionalidades é o grupo que permite ao utilizador realizar todo o processo de modularização de código fonte Java.

Nele é possível criar, alterar e eliminar módulos, atribuir packages aos módulos, definir as propriedades dos módulos e definir as dependências entre módulos.

O processo é sempre que possível, automático e centra-se na janela de modularização.

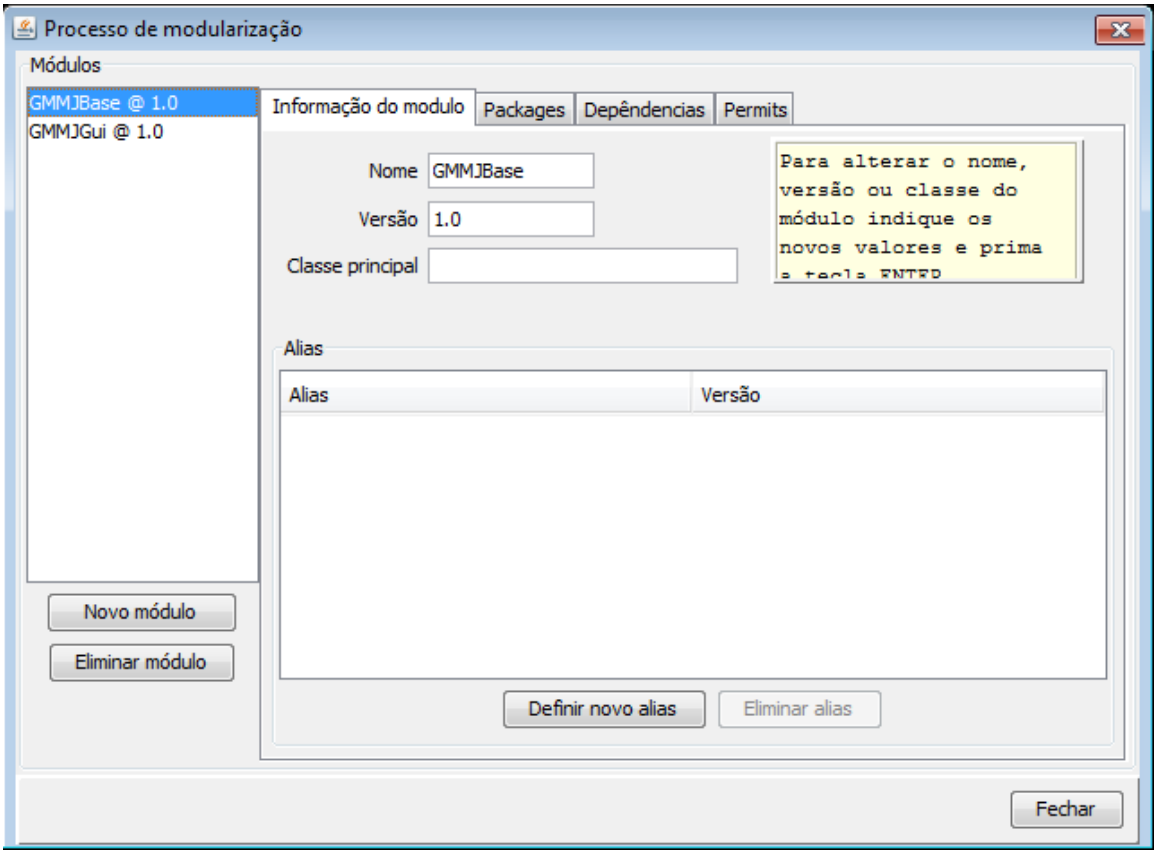


Figura 6.2.5: Janela de gestão do processo de modularização

No processo de definição de módulos, é necessário atribuir os packages aos módulos, processo que deve ser realizado de forma manual através de um interface tal como apresentado na figura seguinte.

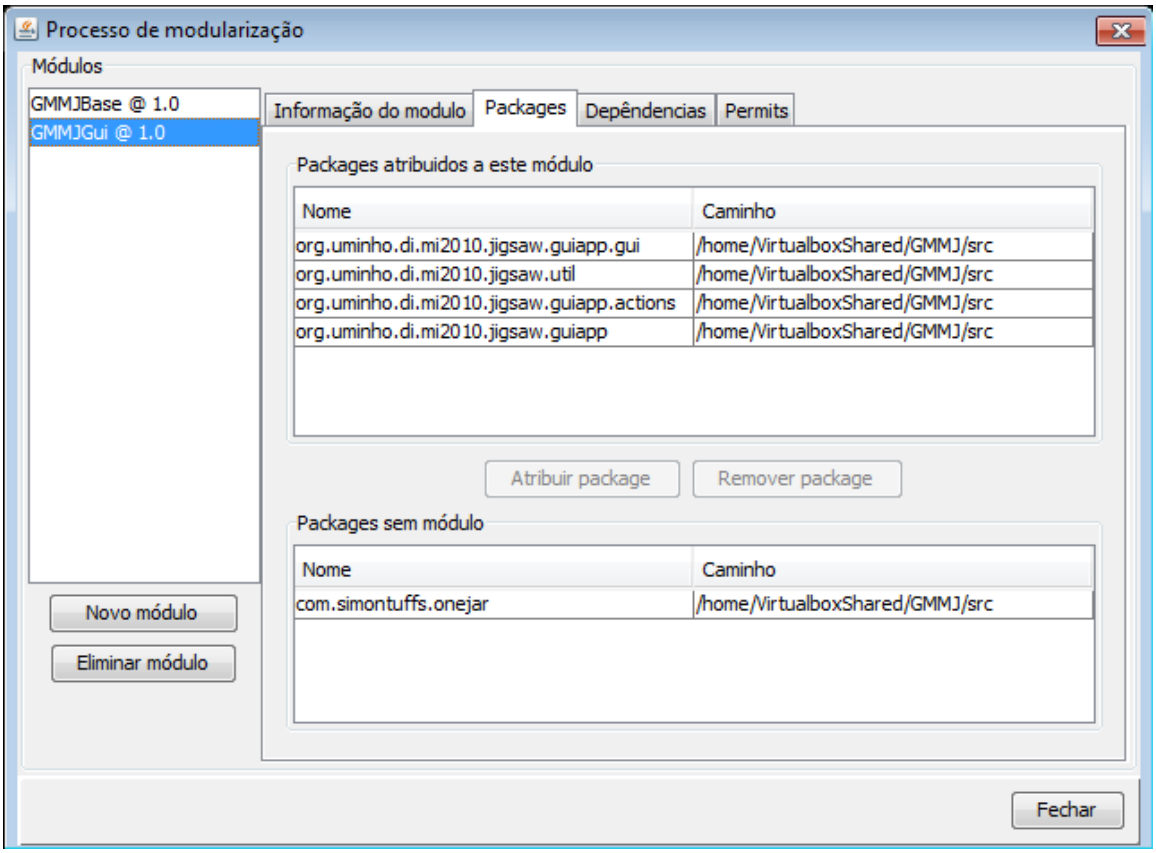


Figura 6.2.6: Modularização - Gestão dos packages atribuídos aos módulos

Após definir que packages fazem parte de cada módulo, é necessário definir as dependências entre módulos, processo que pode ser manual ou automático. A abordagem correta é utilizar sempre o método automático, que procura determinar que módulos, presentes no próprio projeto ou nas bibliotecas importadas podem ser utilizados para satisfazer as dependências do módulo selecionado, tendo como base a análise de código fonte dos packages que constituem cada um dos módulos.

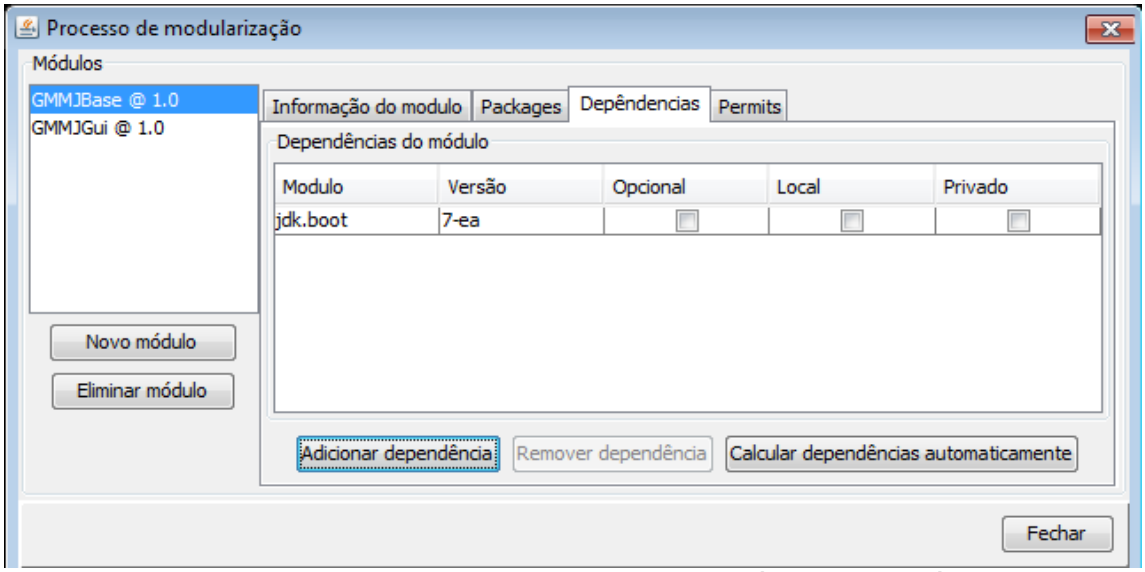


Figura 6.2.7: Modularização - Gestão das dependências dos módulos

No cálculo automático de dependências entre módulos, é aberta uma janela que apresenta todos os requisitos em termos de packages do módulo, que módulos podem satisfazer essas dependências e a biblioteca onde se encontram esses módulos.

Para facilitar o trabalho de modularidade, cada um dos requisitos em termos de dependências é apresentado com um código de cor que permite visualmente identificar o processo de calculo automático de dependências:

- Vermelho: não foi possível encontrar em todas as bibliotecas registadas no projeto um módulo que possa satisfazer a dependência.

- Verde: foi identificado um módulo no próprio projeto ou nas bibliotecas registadas no projeto que permite satisfazer a dependência. O utilizador não tem que fazer nada nesta situação.

- Amarelo: foram identificados mais do que um módulo, no próprio projeto ou em bibliotecas registadas no projeto que permitem satisfazer as dependências. Nesta situação, o utilizar tem que selecionar um entre as várias opções clicando em cima da linha respetiva.

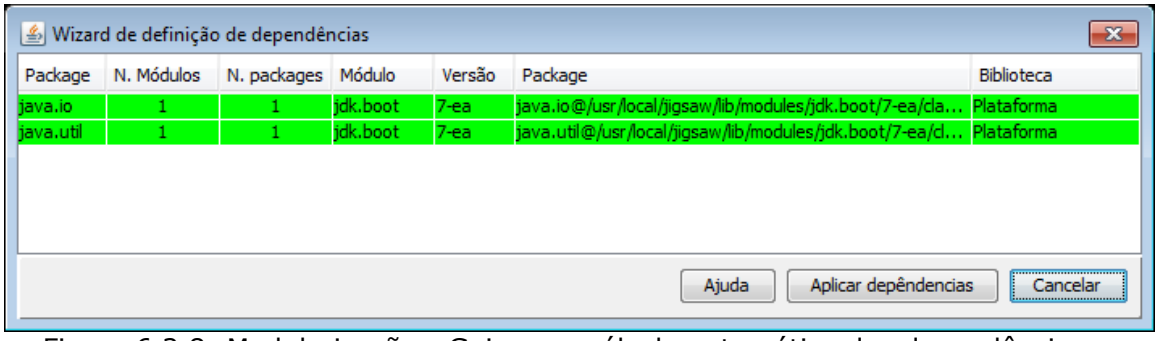


Figura 6.2.8: Modularização - Guia para cálculo automático das dependências

Existem situações em que a metodologia de resolução automática não funciona: i) quando existe mais do que um módulo capaz de satisfazer as dependências; ii) quando não é possível encontrar um módulo que satisfaça as dependências.

No primeiro caso basta escolher um entre os vários módulos possíveis, Esta operação deve ser realizada com atenção e com conhecimento real das dependências.

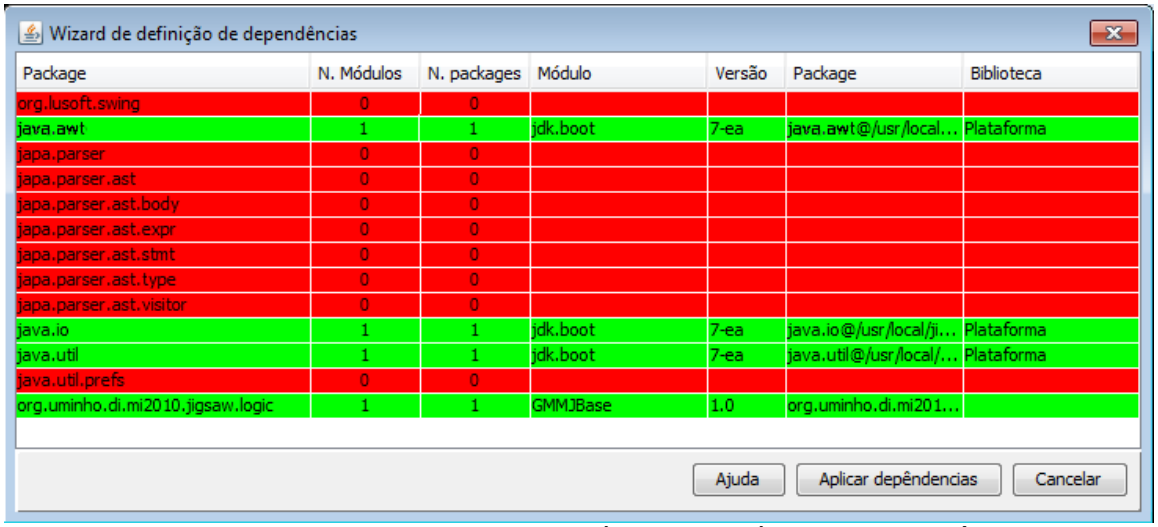


Figura 6.2.9: Modularização - Guia para cálculo automático das dependências

Na segunda situação é necessário definir manualmente um módulo que satisfaça as dependências. Tal funcionalidade consegue-se acedendo à gestão manual de dependências, tal como indicado na figura seguinte.

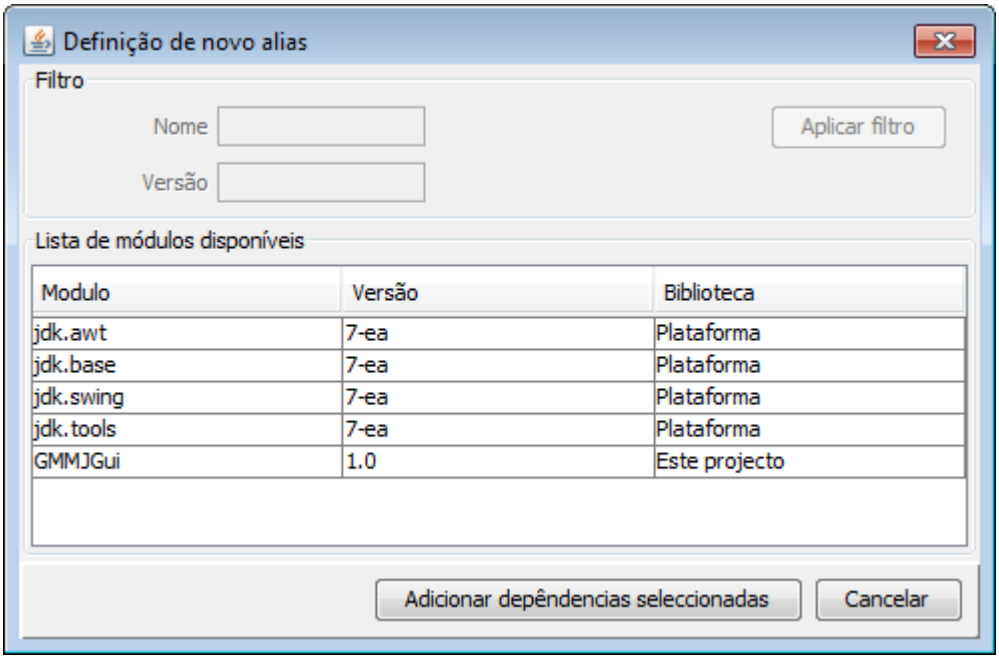


Figura 6.2.10: Modularização - Gestão manual de dependências de módulos

6.2.2.3 Processo de exportação do projeto

Esta funcionalidade permite, após modularização do projeto, exportá-lo e gerar os ficheiros de módulos (module-info.java) para cada um dos módulos, segundo a especificação Jigsaw.

O processo é simples e realizado apenas em um passo. Basta selecionar a opção correspondente no menu "Modularização".

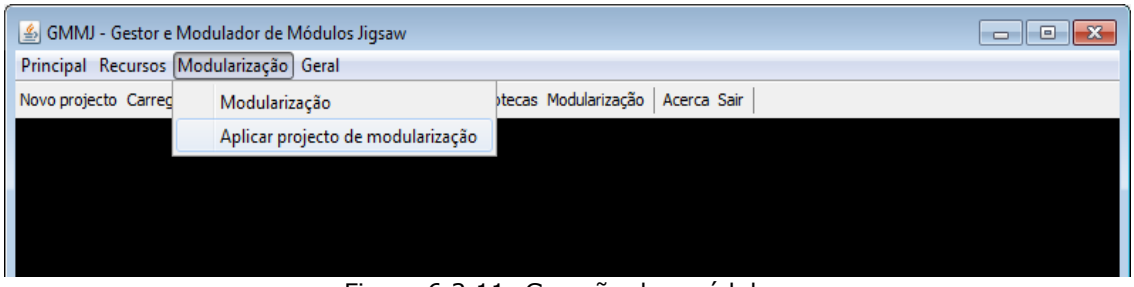


Figura 6.2.11: Geração dos módulos

No fim é indicado o resultado da exportação.

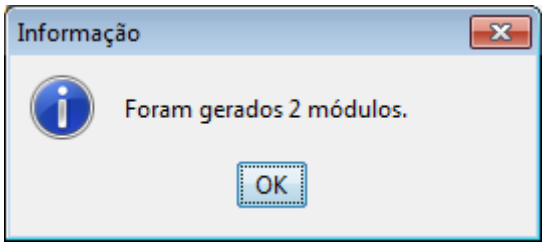


Figura 6.2.12: Resultado da exportação dos módulos

Esta funcionalidade gera os ficheiros "module-info.java" correspondentes a cada um dos módulos exportados.

Os ficheiros são gerados em diretorias com nome com sintaxe "**M@V**", sendo **M** o nome do módulo e **V** a respetiva versão.

Como exemplo e tendo como base os dois módulos que foram definidos para modularizar a aplicação, são gerados 2 ficheiros "module-info.java":

```
module GMMJBase@1.0 {
  requires public jdk.boot@7-ea;
}
```

Tabela 6.2.3: Definição "module-info.java" para o modulo Gaseiforme da aplicação

```
module GMMJGui@1.0 {
  requires public jdk.awt@7-ea;
  requires public jdk.boot@7-ea;
  requires public GMMJBase@1.0;
}
```

Tabela 6.2.4: Definição "module-info.java" para o modulo GMMJGui da aplicação

6.2.2.4 Configurações

Este grupo de funcionalidades disponibiliza ao utilizador um conjunto de parâmetros que podem ser ajustados para definir o comportamento da aplicação, nomeadamente como a aplicação deve funcionar no caso da escrita de ficheiros de forma sobreposta, qual a localização da distribuição Jigsaw.

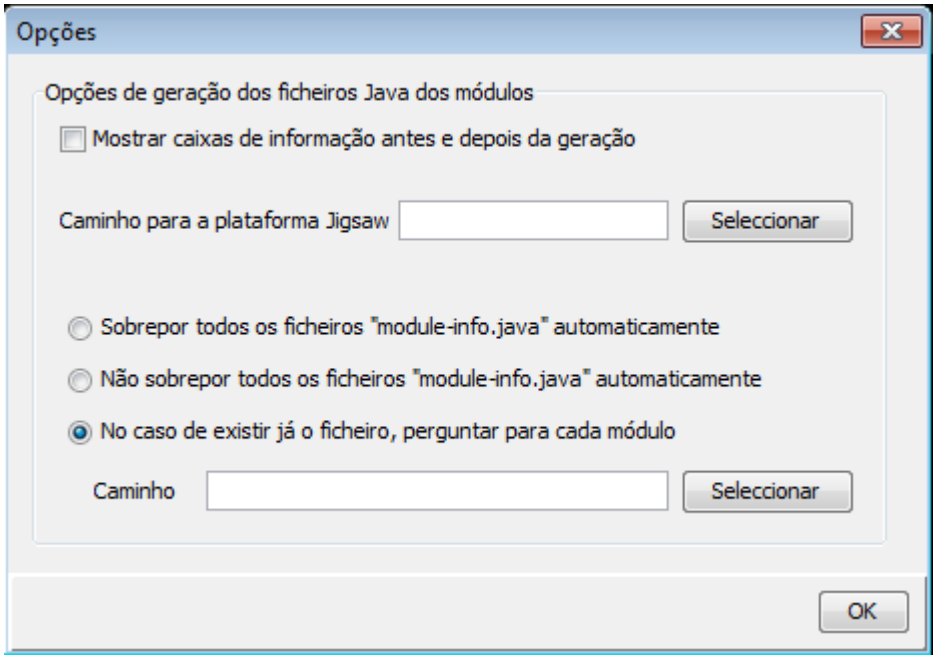


Figura 6.2.13: Caixa de dialogo para configuração das opções da aplicação

6.3 Conclusões

Com a implementação desta aplicação, foram alcançados os objetivos propostos:

- É possível modularizar segundo o modelo Jigsaw, de forma acessível, projetos escritos na linguagem Java.
- É possível, através da aplicação, automatizar a criação de módulos, nomeadamente no que diz respeito ao calculo de dependências.
- É possível gerir bibliotecas apresentadas nesta tese, permitindo que os módulos dessas bibliotecas sejam utilizados para modularizar os projetos do ponto anterior.

No entanto, existem áreas de melhoria que podem enriquecer mais a aplicação:

- Um módulo de apresentação gráfica que permita apresentar todos os módulos e todas as relações entre eles, de forma gráfica, recorrendo ao

modelo de grafos, para uma simples e rápida visualização e entendimento.

- Melhorar a capacidade de deteção de tipos importados pelas classes e interfaces pertencentes ao projeto, permitindo uma melhor capacidade de cálculo de dependências.

- Incorporação da lógica funcional numa ferramenta de linha de comandos que permita a acessibilidade destas funcionalidades (ou da maior parte) a partir de uma linha de comandos.

A atual versão aplicação está pronta a ser utilizada, cumprindo todas as funcionalidades descritas. Não foi no entanto alvo de um extenso processo de testes e correções pelo que sem esse processo, pode ser considerada como sendo uma versão *beta*[29].

29 Versão beta é utilizada geralmente no processo de desenvolvimento de software para descrever o estado de desenvolvimento de uma aplicação que, embora apresentado a maioria das funcionalidades a funcionar corretamente, necessita ainda de um extensivo conjunto de testes e de correção de BUGs.

Capítulo 7

Conclusões

7.1 Introdução

A linguagem Java, desde a sua criação, nunca suportou, de forma direta, na plataforma, uma metodologia modular, nem a própria linguagem teve esse suporte. No entanto, desde as primeiras versões que os arquitetos das aplicações Java adotaram metodologias de desenvolvimento modulares, pois a própria linguagem apelava à utilização dessa metodologia através de conceitos tais como packages e ficheiros JAR. Isto levou a uma diversidade de abordagens, umas mais conseguidas do que outras, sendo que, com o passar dos anos, o número de versões diferentes da plataforma, juntamente como o número de versões que existiam para aplicações e bibliotecas de código, originou o desenvolvimento de formas diferentes de implementar sistemas de modularidade em Java, quase nunca compatíveis entre si e, alguns deles, por sinal os mais utilizados, propícios a problemas graves, nomeadamente colisão de versões e erros de distribuição e operação de aplicações e de causar problemas graves de execução das aplicações, não detetáveis na fase de implementação ou instalação destas, muitas das vezes podendo levar a perdas de dados ou a paragens inesperadas das aplicações por utilização de versões diferentes de bibliotecas de código.

Com esta dissertação, é proposto o estudo da modularidade Jigsaw e de que forma o Jigsaw pode alterar a forma de escrever aplicações Java e de que forma essas alterações podem também ser utilizadas para alterar a forma de escrita de servidores aplicacionais, nomeadamente servidores que já utilizam metodologias

modulares, como por exemplo *plugins*.

7.2 Metodologia modular Jigsaw

Conclui-se que, com a metodologia modular Jigsaw proposta:

- A linguagem Java passa a suportar o conceito de modularidade, na sua especificação, permitindo pensar modularidade logo desde o início da conceção da aplicação.
- A plataforma Java está ela própria implementada segundo a metodologia modular Jigsaw, ou seja, modularizada.
- De forma transversal, o conceito de modularidade é entendido por todas as ferramentas e APIs da plataforma Java.
- A gestão de todo o sistema modular durante a execução das aplicações é assegurada em exclusivo pela máquina virtual Java.
- Passa a ser possível a integração de módulos desenvolvidos por entidades distintas, uma vez que existe apenas uma especificação de modularidade, definida na especificação da linguagem Java e na especificação da máquina virtual Java, que todos devem adotar. Desta forma, deixa de ser necessário ter que optar por sistemas de modularidade incompatíveis entre si, nomeadamente na implementação de sistemas aplicacionais servidores, que pela sua complexidade já utilizam, hoje em dia, sistemas modulares, quer seja através de *plugins* ou de outros modelos.
- Eliminam-se todos os problemas identificados relacionados com as metodologias modulares mais simples, como sejam os ficheiros JAR.
- Melhora-se a integridade de código, dados e de processos, garantindo que problemas recorrentes na plataforma não acontecem.
- Melhoram-se os processos de segurança, garantindo regras de segurança na utilização de módulos logo desde o ato de compilação dos módulos.

- Melhora-se o processo de operação das aplicações, tornando-o mais ágil, eficiente e seguro.

7.3 Aplicabilidade da metodologia Jigsaw no desenvolvimento Java

O estudo demonstrou que é possível implementar aplicações, incluindo servidores de extrema complexidade, recorrendo exclusivamente à metodologia modular Jigsaw.

No entanto, foram identificadas necessidades específicas, existentes por exemplo no desenvolvimento de servidores aplicacionais, que podem também existir noutros tipos de aplicações, que a atual metodologia Jigsaw não prevê pelo que, conclui-se que não é atualmente viável, com base na atual especificação Jigsaw, implementar e integrar a metodologia em servidores aplicacionais para utilização em ambientes reais.

Conclui-se desta forma que, existem alguns cenários no desenvolvimento Java, onde o conceito de modularidade proposto pelo Jigsaw não consegue ter o leque de funcionalidades necessárias para que possa ser utilizado para implementar sistemas modulares em Java.

Conclui-se também que, estas necessidades, encontram-se na maior parte dos casos já implementadas internamente no código do projeto Jigsaw, não estando no entanto publicadas para serem utilizadas externamente ao projeto. Desta forma, trata-se de um problema que facilmente pode ser resolvido tornando então a metodologia Jigsaw a metodologia que pode disponibilizar as aplicações desenvolvidas em Java um verdadeiro ambiente modular. É extremamente importante não considerar estas limitações como impeditivas de que a metodologia possa ser adotada. A sua adoção rapidamente levará a desenvolvimentos que eliminam os constrangimentos identificados e que finalmente fornecerão um verdadeiro conceito de modularidade à plataforma Java.

7.4 Melhorias futuras, o futuro da plataforma Java

Com a aquisição da empresa SUN por parte da Oracle, aquisição esta realizada já depois da escrita desta dissertação ter iniciado, passou a existir uma enorme indefinição relativamente à inclusão da metodologia Jigsaw no JDK 7 ou futuras versões desta plataforma.

A última contribuição de código da equipa de desenvolvimento para o repositório de código do projeto Jigsaw foi no dia 8 de Junho de 2010.

Apesar de várias tentativas de obter informação adicional, através do site oficial do projeto, não foi obtida qualquer resposta pelo que é com algum receio que, dado o aproximar do lançamento da versão 7 do JDK, tudo indique que a metodologia do projeto Jigsaw não venha a ser incorporada na próxima versão da plataforma Java. No entanto, o próprio lançamento da versão da plataforma Java não está definido tendo vindo a ser atrasado nos últimos meses.

Estes múltiplos atrasos podem vir a provocar um grande impacto na plataforma Java e criar divergências importantes entre as empresas e os grupos que dão suporte ao processo JCP e ao avanço da plataforma Java, colocando em risco a importância e notoriedade atingida pela plataforma nas áreas académica e empresarial.

Pode perder-se, neste processo de transição da versão 6 da plataforma para a próxima versão, a oportunidade, única, de dotar a plataforma Java de suporte nativo a um modelo de desenvolvimento modular.

Ainda assim, tendo como base uma enorme base de desenvolvimento realizada, estando muito avançado e perto de uma versão fechada, existem melhorias que podem ser realizadas no modelo Jigsaw ou derivados no futuro. Estas melhorias visam tornar o modelo mais robusto, mais enriquecido do ponto de vista de funcionalidades.

Este dissertação estuda e propõe um conjunto de alterações que visam resolver as limitações identificadas e permitir utilização da metodologia sem

restrições em qualquer cenário de desenvolvimento. A apresentação das melhorias é formal e demonstra a simplicidade das melhorias necessárias para dotar a metodologia de conceitos mais robustos que permitam a utilização em sistemas complexos, como é o caso de servidores aplicacionais.

É fundamental fazer evoluir o modelo:

- Dotar a metodologia de mecanismos de notificação de eventos importantes no processo de gestão de módulos, nomeadamente a capacidade dos módulos observarem esses eventos.

- Definir regras de utilização de repositórios para evitar dispersão de métodos.

- Dotar a máquina virtual Java de capacidade de carregamento e descarregamento de módulos durante a execução das aplicações, de forma programática. Esta funcionalidade é de especial importância para os servidores aplicacionais, mas também pode ser importante para aplicações que são executadas a partir de bibliotecas com repositórios remotos ou que necessitam de oferecer altos níveis de disponibilidade. No conjunto de requisitos da máquina virtual e da especificação de linguagem Java, o que está previsto é que os módulos são definidos estaticamente e que vão sendo carregados na medida em que vão sendo necessários, sem controlo por parte da aplicação, não sendo possível o carregamento de módulos de forma programática.

- Permitir uma melhor e maior integração das aplicações desenvolvidas sem modularidade Jigsaw com a plataforma Java com modularidade Jigsaw. Apesar de se pretender que todas as aplicações passem a ser desenvolvidas tendo como base o modelo apresentado, não devemos nem podemos esquecer a enorme base de aplicações e bibliotecas existentes que foram desenvolvidas em versões anteriores da plataforma Java. Desta forma, deverá ser definida e implementada uma forma de integrar estas aplicações num ambiente modular Jigsaw. Uma possível solução será criar uma camada intermédia, que sirva de interface entre o sistema modular e o software que não suporta módulos, permitindo que o sistema modular aceda ao software não modular através dessa camada.

7.5 Notas finais

Quando iniciei o estudo e escrita desta dissertação, tinha já terminado o Capítulo 3 e tinha já realizado o estudo sobre a metodologia Jigsaw e idealizado o modelo que introduz as alterações defendidas quando a SUN foi adquirida pela Oracle. Ainda assim, não existiram indícios de que o projeto Jigsaw, que serve de base ao estudo, viesse a ser descontinuado. Foi no fim de Julho de 2010 que sinais importantes foram dados de que algo menos positivo para o projeto Jigsaw estava a acontecer quando as listas de contribuição deixarem de ter mensagens novas por parte da equipa de desenvolvimento.

Depois de aproximadamente 6 meses em que o projeto esteve parado, não tendo sido possível qualquer contacto com a equipa que participa no projeto, percebeu-se que existia novamente atividade por parte da equipa de desenvolvimento, no inicio de 2011 (Fevereiro), tendo sido retomado o desenvolvimento.

Existe muita indefinição acerca do futuro do projeto Jigsaw, da sua inclusão na futura versão da plataforma Java e do futuro do conceito de modularidade no Java.

Com esta dissertação espero vir a contribuir, de alguma forma, para que todo o trabalho que existiu até ao momento não caia no esquecimento e que, pelo menos, a ideia de implementar o conceito de modularidade de forma nativa no Java não seja esquecida.

Referências bibliográficas

[TIOBE 10] TIOBE Programming Community, TIOBE Programming Community, 2010, http://www.tiobe.com

[Gosling 05] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Java™ Language Specification, Third Edition, The, Addison Wesley Professional, 2005

[Horstmann 04] Cay S. Horstmann, Gary Cornell, Core Java™ 2 Volume I - Fundamentals, Seventh Edition, Prentice Hall PTR, 2004

[Turner 80] Joshua Turner, Penn Mutual Life, The Structure Of Modular Programs, 1980

[JTML] Sun Microsystems, Inc., Java History, 2010, http://www.java.com/en/javahistory/timeline.jsp

[JCP 10] Sun Microsystems, Inc., Java Community Process, 2010, http://www.jcp.org/

[JCPPROC 09] Sun Microsystems, Inc., JCP 2: Process Document, 2009, http://www.jcp.org/en/procedures/jcp2

[Tate 04] Justin Gehtland, Bruce A. Tate, Better, Faster, Lighter Java, O'Reilly, 2004

[Kiczales 05] Gregor Kiczales, Mira Mezini, Aspect-Oriented Programming and Modular Reasoning, 2005

[Bracha 92] Gilad Bracha, The Programming Language Jigsaw: Mixins, Modularity And Multiple Inheritance, 1992

Strniša 07: Rok Strniša, Peter Sewell, Matthew Parkinson, The Java Module System: Core Design and Semantic Definition,Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications 2007

[JARSPEC] ORACLE, JAR File Specification, 2010, http://download.oracle.com/javase/6/docs/technotes/guides/jar/jar.html

[Crawford 05] William Crawford, Jim Farley, Java Enterprise in a Nutshell, 3rd Edition, O'Reilly, 2005

[Darwin 04] Ian F. Darwin, Java Cookbook, 2nd Edition, O'Reilly, 2004

[JSR277] Sun Microsystems, Inc., JSR-277: Java™ Module System, 2007, http://jcp.org/en/jsr/detail?id=277

[OSGI] OSGi Alliance, OSGi Alliance, 2010, http://www.osgi.org

[JSR291] OSGI Alliance, JSR 291: Dynamic Component Support for JavaTM SE, 2006, http://jcp.org/en/jsr/detail?id=291

[ECLIPSE] Eclipse Foundation, Eclipse Project, 2011, http://www.eclipse.org

[Ahn 06] Heejune Ahn, Nowon-gu, Chang Oan Sung, Towards Reliable OSGi Framework and Applications,

[OSGi 09] OSGI Alliance, OSGi Service Platform Core Specification, OSGI Alliance, 2009

[Geoffray 08] Nicolas Geoffray, Gaël Thomas, Charles Clément, Towards a new Isolation Abstraction for OSGi, 2008

[JSR294] Sun Microsystems, Inc., JSR-294: Improved Modularity Support in the Java™ Programming Language, 2006, http://jcp.org/en/jsr/detail?id=294

[NETBEANS] Projecto Netbeans, Projecto Netbeans, 2010, http://www.netbeans.org

[Bernardi 09] Mario Luca Bernardi, Giuseppe Antonio Di Lucca, A Role-based

Crosscutting Concerns Mining Approach toEvolve Java Systems Towards AOP, 2009

[Cazzola 05] Walter Cazzola, Sonia Pini, Massimo Ancona, AOP for Software Evolution: A Design Oriented Approach, 2005

[Kiczales 01] Gregor Kiczales, Erik Hilsdale,Jim Hugunin, Mik Kersten,Jeffrey Palm, and William G. Griswold, GETTING STARTEDWITH ASPECTJ, 2001

[ASPECTJ] Projecto AspectJ, Projecto AspectJ, 2010, http://www.eclipse.org/aspectj

[JIGSAW] Sun Microsystems, Inc., OpenJDK Jigsaw Project, 2010, http://openjdk.java.net/projects/jigsaw

[MYSQL 10] MySQL, MySQL, 2010, http://www.mysql.org

[TOMCAT 10] Apache Software Foundation, Apache Tomcat, 2010, http://tomcat.apache.org

[Kassem 00] Nicholas Kassem, Designing Enterprise Applicationswith the JavaTM 2 Platform, Enterprise Edition, Sun Microsystems, Inc., 2000

[Johnson 09] Javid Jamae, Peter Johnson, JBoss in Action, Manning Publications Co., 2009

[Marchioni 09] Francesco Marchioni, JBoss AS 5 Development, Packt Publishing, Ltd., 2009