Departamento de Ciências e Tecnologias de Informação

Modelo de internacionalização em bases de dados

Catarina Anunciação Costa Dias dos Santos

Dissertação submetida como requisito parcial para obtenção do grau de

Mestre em Software de Código Aberto

Orientador:

Professor Doutor Sancho Moura Oliveira,

Professor Auxiliar do Departamento de Ciências e Tecnologias da Informação do

ISCTE-IUL

Setembro, 2015

Agradecimentos

Finalizando a elaboração desta dissertação, a qual permite obter grau de Mestre em Software de Código Aberto, quero agradecer a quem de alguma forma contribuiu para que tal se tornasse realidade.

Antes de mais, quero apresentar os meus agradecimentos ao Professor Doutor Sancho Oliveira por me acompanhar e orientar nesta etapa, pelos contributos e sugestões que foram sempre uma mais-valia para o enriquecimento deste trabalho.

Quero também agradecer à Professora Doutora Manuela Aparício e Professor Doutor Carlos Costa, pela disponibilidade em realizar reuniões periódicas de esclarecimento de dúvidas com os alunos do Mestrado em Software de Código Aberto (MOSS) e terem, assim, auxiliado também na definição da estrutura de investigação desta dissertação.

Não posso deixar de agradecer, também, aos meus colegas do MOSS, Gaspar Brogueira e Carlos Bernardino, pelas ideias partilhadas, pelo incentivo que me deram ao longo desta caminhada e pelo facto de terem sido a minha ponte de ligação às aulas do MOSS, durante os longos períodos de tempo em que estive em Angola. Neste aspeto, aproveito também para agradecer a tolerância por parte dos professores MOSS em geral, a qual me permitiu apresentar trabalhos e fazer testes fora dos dias inicialmente estipulados.

Quero deixar aqui um enorme obrigado aos meus pais, por acreditarem em mim desde o primeiro dia, por todo o apoio e pelo carinho demonstrado. Enfim, não há nada que pague aquele Tupperware de comida caseira que chega de vez em quando, para poupar o tempo de fazer o jantar.

Quero também agradecer aos meus avós, irmãos, primos e amigos mais próximos, em especial Carla Scarpa, por acreditarem em mim, até mesmo quando eu própria duvidava, pela coragem transmitida e por me proporcionarem momentos de boa disposição ao longo desta caminhada.

Por fim, mas não menos importante, a força de vontade que ressurgiu sempre a cada momento de desânimo, essa devo-a agradecer a Deus.

Resumo

A presente proposta de dissertação de mestrado enquadra-se no âmbito da localização de *software*. Esta tem como principal objetivo definir um modelo de internacionalização para bases de dados. Este modelo tem em vista habilitar a internacionalização numa base de dados facilitando, assim, o trabalho tanto do programador de base de dados como do tradutor que efetua a sua localização. O uso deste deverá ser o mais simples e direto possível para o programador e transparente para o tradutor, não sendo necessária a indisponibilização da base de dados para manutenção pós-produção destas mensagens.

Neste sentido, foi desenvolvido um módulo, interno ao motor de base de dados, que permite separar todos os elementos dependentes do idioma em uso. Esta separação torna a fonte de elementos independente o suficiente para que os seus elementos possam ser editados, por uma pessoa sem qualquer conhecimento em bases de dados, e recarregados em tempo de execução, isto é, com a base de dados em total funcionamento.

Este modelo pode ser útil em qualquer tipo de aplicação, sendo a sua utilidade mais evidente no desenvolvimento de aplicações empresariais, nomeadamente, aplicações com grande parte da sua lógica de negócio aplicada ao nível da base de dados. Por lógica de negócio entende-se: regras e validações que definem o negócio da empresa. Com esta solução, não é necessário criar modelos de dados específicos para internacionalização, aos quais estão associados métodos de pesquisa e manutenção mais dispendiosos.

Palavras-chave

Localização de *software*, base de dados, modelo de internacionalização.

Abstract

The proposed master’s dissertation falls within the scope of software localization. This has as main objective to define an internationalization model for databases. This model is intended to enable the internationalization in a database, thus facilitating the work of both the database programmer as the translator which performs its localization. The use of this should be as simple and direct as possible to the developer and transparent to the translator, not being necessary to unavailable database for post-production maintenance of these messages.

Accordingly, it was developed an internal module to the database engine, which allows the separation of all the elements dependent on the language being used. The separation makes the elements’ source independent enough so that its elements can be edited by a person without any databases knowledge and refilled at run time, i.e. with the database in full operation.

This model may be useful in any type of application being its most evident utility in the development of business applications, particularly applications with much of its business logic applied to the database level. By business logic means: rules and validations that define the company's business. With this solution, it is not necessary to create specific data models for internationalization, which are associated with more expensive methods of research and maintenance.

Keywords

Software localization, database, internationalization model.

Índice

Índice de Tabelas

Índice de figuras

Lista de Abreviaturas

ASCII – Código padrão americano para troca de informação *(American Standard Code for Information Interchange)*

CRUD – Quatro funções básicas SQL *(Create, read, update, delete)*

CSV – Valores separados por vírgula *(Comma Separated Values)*

DB – Base de dados *(Database)*

DTP – Editoração eletrónica *(Desktop Publishing)*

FOSS – Software de código aberto e livre *(Free Open Source Software)*

GUI – Interface gráfica do utilizador *(Graphical User Interface)*

I18N – Internacionalização *(Internationalization)*

IDE – Ambiente de desenvolvimento integrado *(Integrated Development Environment)*

ISO – Organização internacional de padronização *(International Standard Organization)*

JDBC – Conectividade de base de dados Java *(Java Database Connectivity)*

JVM – Máquina Virtual Java *(Java Virtual Machine)*

L10N – Localização *(Localization)*

MOSS – Mestrado em Software de Código Aberto *(Master in Open Source Software)*

MVC – Padrão arquitetural de *software* Modelo-Vista-Controlador *(Model-View-Controller)*

OOo – OpenOffice.org

QA – Garantia de qualidade *(Quality Assurance)*

RTF – Formato de texto rico *(Rich Text Format)*

SGBD – Sistema Gestor de Base de Dados

SQL – Linguagem de consulta estruturada *(Structured Query Language)*

UI – Interface do utilizador *(User Interface)*

UTC – Coordenada de tempo universal *(Universal Time Coordinate)*

UTF – Formatos de transformação Unicode *(Unicode Transformation Formats)*

XML – Linguagem extensível de marcação genérica *(Extensible Markup Language)*

ZF – Zend Framework

1 Introdução

A localização de *software* torna um produto, um *software* e tudo relacionado a ele, local, isto é, adaptado ao idioma e às convenções locais. Portanto, localização é o processo que adapta o produto ao mercado local (Prudêncio, Valois, & Lucca, 2004). Dividir este processo em etapas favorece a gestão e torna esta complexa tarefa administrável, uma vez que esta supõe um fluxo de trabalho complexo e cuidadoso, para a obtenção de um produto final de qualidade. As principais etapas deste processo costumam ser constituídas por análise do material original e levantamento terminológico dos termos utilizados na interface de utilizador e na documentação e aprovação por parte do cliente dos termos traduzidos. Posteriormente, parte-se para a localização da interface em si, isto é, menus, botões, caixas de diálogo, entre outros. De seguida é a vez da tradução da documentação com base na terminologia pré-aprovada já utilizada na interface. Por fim, surgem as fases de revisão técnica do material traduzido, testes e controlo de qualidade pré-entrega (Prudêncio et al., 2004).

No seguimento desta secção introdutória serão apresentados o problema, os objetivos de investigação e respetiva metodologia, estrutura da dissertação e ainda as contribuições da mesma.

1.1 Definição do problema e objetivos

Os utilizadores pelo mundo inteiro esperam sempre que um *software* “fale” com eles na sua língua. Isto não é uma questão de orgulho nacional, mas sim uma questão de produtividade. Utilizadores que compreendam o produto na sua totalidade serão mais hábeis e capazes de evitar erros, assim sendo, vão sempre preferir aplicações na sua língua e adaptadas ao seu ambiente cultural (Hau & Aparício, 2008).

Hoje em dia cada vez são mais as aplicações que são usadas em vários países de diferentes línguas. No passado, a maioria dos programas de *software* só podiam "falar uma língua". Por exemplo, o *software* desenvolvido no Reino Unido apenas poderia falar inglês, enquanto o mesmo software desenvolvido na China só poderia falar chinês. Portanto, era necessário desenvolver várias versões do mesmo programa em que divergia, por exemplo, apenas a língua. A crescente globalização veio permitir que um

determinado produto pudesse ser comercializado em vários países (Darcy G. Benoit, 2004). Esta realidade incontornável, obrigou a que o desenvolvimento de *software* contemplasse políticas relacionadas com a globalização e internacionalização (Batista, 2014).

Para muitas empresas, que tencionam lançar os seus próprios produtos de *software* em diferentes mercados, a internacionalização e a localização podem tornar-se processos árduos e caros (McCollough, 2011). Em função da complexidade, a localização é quase sempre terceirizada para empresas especializadas, pois distancia-se das funções básicas das empresas de desenvolvimento de *software* e não é uma atividade realizada diariamente para justificar uma estrutura interna fixa para tal. No entanto, não se pode descuidar em nenhum momento desta atividade, uma vez que a interface e a funcionalidade adequadas do produto representam a diferença entre conquistar um mercado ou perdê-lo por não ter em consideração o seu idioma ou convenções locais (Prudêncio et al., 2004).

Atualmente, alguns dos ambientes de desenvolvimento existentes, já integram um modelo de internacionalização, o qual permite criar com facilidade sistemas de informação que devem suportar várias línguas. Uma aplicação bem-sucedida, usando o módulo de internacionalização, parecerá ter sido desenvolvida exclusivamente para o idioma e/ou local em questão (Oracle, 2011). Quer isto dizer que uma aplicação *web* executada num *browser* em inglês que parece ser desenvolvida especificamente na língua inglesa e, ao mesmo tempo, esta mesma aplicação executada num *browser* em francês parece ser desenvolvida especificamente em francês, então esta é uma aplicação bem-sucedida no que toca a internacionalização.

Normalmente este tipo de modelos são implementados usando ficheiros de propriedades – um por cada linguagem suportada – que contêm pares de chave-valor. A chave é referenciada no código da aplicação onde quer que um elemento de texto padronizado seja necessário. O valor corresponde ao texto, na linguagem específica, associado à chave referenciada. A linguagem a ser apresentada é gerida internamente no código (Jellema, 2012).

Atualmente ainda não existe este tipo de modelos ao nível das bases de dados, a não ser aproximações através da criação de modelos relacionais desenvolvidos em específico para o efeito. Esta limitação dificulta bastante o processo de localização do *software*, principalmente se a lógica de negócio for predominante ao nível da base de dados, isto

é, se as regras que definem o negócio da empresa forem predominantes na camada de dados do *software*. Da aplicação destas regras resulta informação que é guardada na própria base de dados, sendo disponibilizada posteriormente ao utilizador via *front end*. Desta forma, o ideal seria a própria base de dados ter já disponível no seu *core* um modelo de internacionalização operacional e prático.

Neste âmbito, o trabalho desenvolvido nesta dissertação pretende responder à seguinte questão:

- *Como integrar um modelo de internacionalização em Bases de Dados pré-existentes?*

Desta forma, o que se pretende com esta dissertação, enquadrada no Mestrado em Software de Código Aberto, é estudar uma forma de integrar um modelo de internacionalização numa base de dados, tendo os seguintes pontos como objetivos gerais de investigação:

- Identificar o estado da arte relativamente a outros sistemas de internacionalização.
- Elaborar um desenho de visão da solução refletindo o modelo conceptual da mesma.
- Desenvolver um protótipo do modelo numa base de dados *open source*.
- Validar a integração do módulo de internacionalização (I18n), resultante do protótipo anterior, em bases de dados fechadas.

1.2 Metodologia

A metodologia seguida para o desenvolvimento da dissertação começou pela revisão da literatura sobre o estado da arte relativamente a outros sistemas de internacionalização no sentido de perceber como os mesmos estão desenhados.

Findo este enquadramento, seguiu-se a elaboração de um desenho de visão do modelo a ser desenvolvido definindo, assim, o modelo conceptual da solução. Posteriormente, foi desenvolvido um protótipo deste modelo numa base de dados *open source*, com o objetivo de este ser parte integrante do *core* da respetiva base de dados, tendo como suporte investigação tecnológica e atividades experimentais.

Por fim, foi analisada a possibilidade de integração do módulo I18n, resultante do protótipo anterior, em bases de dados fechadas, no sentido de perceber até que ponto este módulo é genérico.

De modo a clarificar esta parte, segue abaixo uma tabela com indicação da metodologia adotada para cada um dos objetivos de investigação desta dissertação.

Tabela 1: Objetivos e metodologias associadas

Objetivo	Método
Identificar o estado da arte.	Efetuar revisão de literatura, com base em artigos, fóruns e *websites* sobre internacionalização, localização e desenhos de sistemas I18n já existentes.
Elaborar desenho de visão da solução refletindo o modelo conceptual.	Analisar de forma crítica, alto nível, as várias possibilidades para a nova solução elaborando um esquema de representação da mesma.
Desenvolver protótipo em base de dados de código aberto.	Obter o código fonte de um motor de base de dados *open source* e estudar a sua estrutura interna. Adquirir *know how* com investigação tecnológica, acompanhada de *debugging* do código fonte. Realizar os desenvolvimentos necessários e registo dos mesmos, com o objetivo de chegar ao modelo definido.
Validar integração em bases de dados de código fechado.	Estudar, com base em investigação tecnológica e atividades experimentais, a possibilidade de integrar, numa base de dados fechada, o módulo I18n resultante do modelo desenvolvido no ponto anterior.

1.3 Estrutura da Dissertação

Esta dissertação divide-se em cinco partes, em que a primeira é esta parte introdutória. Na segunda parte encontra-se a revisão de literatura onde é feito o levantamento do estado da arte relativamente a internacionalização, localização e sistemas I18n já existentes. Na terceira parte, tendo a revisão de literatura como suporte, é elaborado um desenho de visão da solução definindo, assim, o modelo conceptual da mesma. Na quarta parte inicia-se a parte empírica desta dissertação, onde é desenvolvido o trabalho de investigação tecnológica, acompanhado de atividades experimentais e respetivos resultados, para a criação de um protótipo funcional do modelo projetado. Ainda nesta parte é também investigada e testada uma forma de integração do módulo I18n desenvolvido numa base de dados de código fechado. Na quinta e última parte encontra-se a conclusão dos resultados obtidos, limitações e sugestões de trabalhos futuros.

De forma a ajudar a visualizar esta estrutura de investigação, foi elaborada a seguinte tabela:

Tabela 2: Estrutura de Investigação

<table>
<tr><th colspan="5">Desenho de Investigação</th></tr>
<tr><td>2ª Parte</td><td>3ª Parte</td><td colspan="3">4ª Parte</td></tr>
<tr><td>Revisão de Literatura</td><td>Modelo Conceptual</td><td colspan="3">Protótipo</td></tr>
<tr><td>Estado da Arte</td><td>Desenho/Visão da solução</td><td>Investigação tecnológica</td><td>Atividades experimentais</td><td>Integração em base de dados fechada</td></tr>
<tr><td></td><td></td><td colspan="3">Trabalho Empírico</td></tr>
</table>

Na tabela acima são apresentadas apenas a segunda, terceira e quarta parte, pois a primeira é esta parte introdutória de apresentação da dissertação, onde se definem os objetivos e metodologias de investigação, precedidos pela questão de investigação. Por outro lado, a quinta e última é referente à conclusão da dissertação, onde se apresentam os resultados obtidos e os trabalhos futuros.

A execução das diversas tarefas apresentadas nesta tabela, não obedecem a esta ordem exatamente, pois algumas delas têm que ser executadas em simultâneo.

1.4 Contribuições e Publicações

A presente dissertação pretende contribuir, teoricamente, com um modelo conceptual para internacionalização em bases de dados. Modelo este que será concretizado com um protótipo desenvolvido numa base de dados *open source*. Desta forma, a contribuição teórica desta dissertação é o modelo conceptual definido e, por outro lado, a contribuição prática é o protótipo desenvolvido para concretização do respetivo modelo.

2 Revisão da Literatura

Para o desenvolvimento da dissertação recorreu-se à revisão da literatura sobre o que é internacionalização e localização de *software* (L10n). A globalização, no contexto dos sistemas de informação, abarca os conceitos de internacionalização e localização (Batista, 2014).

Adicionalmente, efetuou-se também revisão de literatura sobre alguns sistemas I18N já existentes no sentido de identificar o estado da arte.

2.1 Internacionalização de *software*

"Internationalization is not a feature!" (Gillam, 1999)

A Internacionalização é um processo de engenharia de *software* cujo principal objetivo, é torna-lo flexível e neutro em termos de relações culturais, financeiras e legais de um país, isto é, projetar um produto que pode ser adaptado a diferentes culturas. O processo I18n deve começar logo no início do processo de desenvolvimento do *software*, aquando da identificação dos requisitos, apesar de a internacionalização não ser considerada uma funcionalidade, pois os clientes não têm por hábito referir que querem uma aplicação internacionalizada, mas apenas partem do princípio que esta irá funcionar corretamente para a língua deles (Gillam, 1999). O objetivo da I18n é garantir que o *software* possui uma arquitetura computacional de base que permita a localização sendo, assim, separados os elementos dependentes da linguagem ou da cultura, caso contrário, o *software* terá que ser reescrito a partir do zero (Batista, 2014). Esta envolve a preparação de código fonte original para o processo de localização, sendo vários os pontos que devem ser considerados, tais como (Aragão, 2004):

- Esquemas de cores e escolhas de gráficos que evitem ofensas a potenciais clientes.
- Caixas de diálogo largas o bastante para acomodar a expansão de texto.
- Funcionalidades que suportem vários formatos de data, hora e moeda.

- Funcionalidades de *input* e *output* que suportem vários *character sets*, inclusive de dois bytes (mercado asiático).
- Campos de texto justificados para prevenir que o texto expandido em outras linguagens se sobreponha aos gráficos.
- Interface de utilizador facilmente adaptável para permitir a clientes de várias origens lerem textos tanto da esquerda para a direita como da direita para a esquerda.

Na fase de implementação de *software*, devem ser identificados os componentes que têm que se ajustar a um determinado local/região. Estes componentes devem ser separados do núcleo do programa sendo, assim, o *software* dividido estruturalmente em duas grandes áreas: código fonte (*source code*) e ficheiros de recursos (*resource files*). O código fonte, corresponde ao núcleo do programa, ou seja, a parte que não irá sofrer alterações independentemente do local onde o *software* é instalado. Os chamados ficheiros de recursos, compreende todos os componentes que serão ajustados a cada local/região (Batista, 2014). Os tradutores, enquanto peritos em falar e escrever um idioma estrangeiro, não são necessariamente engenheiros de *software*. Separando o texto do código, aumenta-se a segurança para o correto funcionamento das engrenagens vitais do produto. O principal senão desta separação é o fato do tradutor não ser capaz de ver e entender o contexto no qual os textos se inserem. A falta de informação contextual pode acarretar em traduções dúbias e errôneas. Para evitar este problema, o ficheiro traduzido deve ser reinserido na aplicação e compilado de modo que o tradutor possa verificar a UI completada com a informação contextual traduzida (Aragão, 2004). Por outro lado, ainda no seguimento deste tema, o uso de linhas de comentário nos ficheiros de textos é sempre uma mais-valia para contextualizar o tradutor.

2.2 Localização de *software*

O desenvolvimento de *software* segundo a norma I18n, permite a execução do processo de localização (Batista, 2014). Desta forma, a L10n é aplicada a *software* codificado segundo a norma I18n e consiste em transformar o idioma de origem do *software* num ou mais idiomas de destino, dando ao produto o aspeto de ter sido criado no país designado, isto é, consiste no ajustamento de todos os ficheiros de recursos a um determinado país/local não sendo necessárias alterações ao código fonte (Batista, 2014).

Este processo possui duas dimensões fundamentais: localização de texto e localização cultural, sendo que a primeira consiste na tradução da interface e documentação da aplicação e a segunda compreende o respeito de todas as convenções culturais de uma determinada região (Batista, 2014).

Relativamente ao que se deve localizar, em muitos casos, o que o define é o orçamento ou o prazo para a localização. Por outro lado, a decisão de quantas línguas a serem traduzidas é frequentemente derivada da demanda do mercado e/ou das exigências legais (Aragão, 2004).

O localizador é um especialista em tradução na área de *software*, com sólidos conhecimentos do idioma em que o *software* está escrito (idioma de partida) e perfeito domínio do idioma para o qual irá traduzir (idioma-alvo), sendo este o idioma nativo do localizador. É particularmente importante o uso de indivíduos no pais-alvo em vez de indivíduos que já lá viveram ou trabalharam, porque, apesar de muitos aspetos do conhecimento do país serem quase imutáveis (linguagem), por exemplo, o significado das cores é mais sujeito a mudanças culturais. Desta forma, um individuo que não seja nativo do pais-alvo pode não estar atualizado sobre estas pequenas mudanças de significado (Batista, 2014).

O localizador deve ser o responsável pela tradução de conteúdo eletrónico (*websites*, textos de ajuda *on-line*, a interface de *software* – botões, menus, janelas), da documentação técnica de *software* e material colateral (material de marketing, embalagens de produtos, folhetos de referência rápida, documentação de registo, formações baseadas em computador, entre outros), inclusive gráficos e multimédia. Também se podem atribuir ao localizador as atividades complementares ligadas às questões linguísticas, como a redação técnica de manuais, revisão dos textos traduzidos (GUI – interfaces gráficas, ajuda, manuais), levantamento e tradução da terminologia utilizada no *software*, criação/manutenção de glossários de termos técnicos, entre outros (Prudêncio et al., 2004).

A comunicação com os utilizadores não é feita apenas com texto, mas também com gráficos, sinais e ícones. Existem numerosos sinais (por exemplo, de erro ou perigo) e ícones com significados especiais cujos utilizadores devem estar familiarizados com o seu significado. Por isso, um dos principais desafios do ajustamento dos ficheiros de recurso de um determinado local/país é ajustar os elementos gráficos do *software* (cores, símbolos, sinais), direccionalidade do texto (se aplicável), fonte utilizada nos textos e

ajustamento do texto ao *layout* gráfico. Os componentes comummente afetados são os menus, mensagens, caixas de diálogo, imagens, sons, barras de ferramentas e barras de estados (Batista, 2014).

O sucesso do processo de localização depende muito das etapas de globalização e internacionalização do *software*, realizadas pelo fabricante. Estas etapas estratégicas visam o planeamento da comercialização do *software* em outros países (globalização) e o desenvolvimento do mesmo voltado para a localização, para que ele já seja desenvolvido de forma a facilitar o processo (internacionalização). Uma das principais dificuldades no processo de localização é ter acesso ao texto falado ou escrito para traduzi-lo e depois inseri-lo de volta no produto. Para que essa tarefa seja possível, a engenharia é uma etapa fundamental. Dentre outras tarefas mais específicas, os engenheiros de *software* extraem todo o texto traduzível e convertem os arquivos específicos do *software* em formatos mais acessíveis para os tradutores, como por exemplo, ficheiros em formato de texto rico (RTF). Estes ficheiros costumam ser usados para intercâmbio de documentos entre diversas plataformas, pois a maioria dos processadores de texto são capazes de ler e escrever documentos RTF. Em geral, são esses arquivos que são enviados para os gestores de projeto nas empresas de cada país (Ribeiro, 2005).

Com base na interpretação dos conceitos de projeto e de gestão de projetos descreve-se o processo de gestão de projetos de tradução (Resende & Silva, 2009). Ao gestor de projeto cabem as tarefas de elaborar o cronograma do projeto, considerando sempre o prazo estabelecido pelo cliente; selecionar a equipa; organizar o material enviado e enviar os arquivos para a engenharia, além de fazer todo o acompanhamento e a conclusão do projeto (Ribeiro, 2005).

A etapa seguinte à tradução é a revisão. Nesta fase, profissionais em geral mais experientes no processo e na área de especialidade do *software* (informática, finanças ou medicina, por exemplo) comparam o texto original e o traduzido, verificando a precisão da tradução em relação ao original, o estilo e a correção gramatical e a padronização terminológica do texto traduzido. Após a revisão, temos a etapa de *proofreading*, de leitura final do material, sem referência ao texto original, apenas para garantir a coerência e a correção gramatical do texto final. Em seguida, os arquivos são enviados para a engenharia para verificação e para atualização da memória de tradução. Após a compilação, começa a fase de teste do *software*, em que são verificados a visualização e

o funcionamento de todas as opções, caixas de diálogo e mensagens. Esta é uma etapa crucial para o aspeto visual do processo de localização: é essencial que todos os componentes possam ser corretamente visualizados pelo utilizador e que os recursos do *software* estejam a funcionar corretamente na versão localizada. Em relação ao material impresso (normalmente o guia do utilizador e a embalagem), a etapa final é a editoração eletrónica, ou DTP (*Desktop Publishing*), para formatação final. Após a conclusão de todas estas etapas, o projeto é submetido a um processo de controlo de qualidade (QA, *Quality Assurance*), cujas etapas podem variar de empresa para empresa (Ribeiro, 2005).

Com base no descrito anteriormente, conclui-se que os processos I18n e L10n são processos em que muitas das tarefas são feitas por seres humanos, portanto, recomenda-se a colaboração entre os elementos das várias equipas (programadores, tradutores, utilizadores) intervenientes nestes processos (Gross, 2006). A Figura 1 mostra de relance como estas equipas podem colaborar entre si.

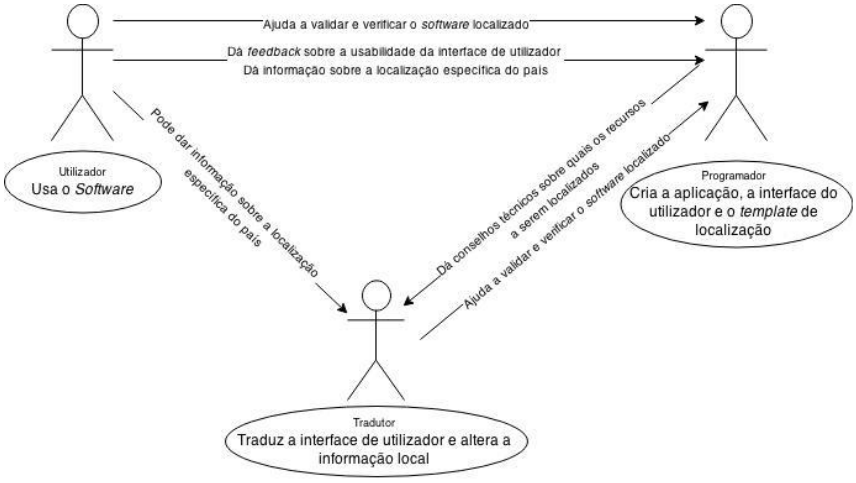


Figura 1 - Colaboração entre os elementos das várias equipas durante o processo I18n e L10n (Adaptado de Gross, 2006)

Relativamente ao *software* de código aberto e livre (FOSS), a localização não é muito diferente da localização de *software* comercial. Excetuando o facto de que este é localizado tipicamente por voluntários que trabalham remotamente, não tendo o suporte de tradutores especializados nem de dicionários de termos técnicos. Neste caso, estes processos são apoiados por sistemas que mantêm histórico. Histórico este que ajuda os utilizadores a acompanhar todas as alterações efetuadas aos ficheiros, assim como o respetivo autor. A cada tradução publicada no repositório por um voluntário existe a respetiva validação por parte da restante comunidade, que pode editar e fazer correções até o processo atingir a máxima qualidade (Batista, 2014).

2.3 Formatos de codificação

A necessidade de codificação eletrónica surgiu pela primeira vez quando as pessoas tentavam enviar mensagens através de linhas telegráficas, utilizando o código Morse[1]. Este é considerado o início da codificação de caracteres, contudo, é diferente da forma como os computadores modernos codificam a informação (McEnery & Xiao, 2005).

O avanço posterior na codificação de caracteres foi o código Baudot (1874) de 5-bit, o qual consegue codificar apenas 32 caracteres por nível e, utilizando um esquema de alternância (equivalente à tecla "SHIFT" do teclado), duplica a sua capacidade de codificação alternando entre os dois níveis de 32 caracteres. Este formato já foi útil para o texto ser devidamente apresentado e impresso, pois para além de suportar os principais caracteres alfanuméricos e pontuação suporta, também, caracteres de controlo, tais como, sinais de início de texto, fim de texto e reconhecimento (McEnery & Xiao, 2005).

Uma desvantagem do código de Baudot é este não permitir o acesso aleatório a um caracter específico da cadeia de caracteres, pois é necessário um *bit* adicional para o efeito. Desta forma, surgiu a codificação de caracteres de 6-bit, inventada pelo americano Herman Hollerith (1860-1929), suportando já 69 caracteres e que foi amplamente usado até à década de 1960 (McEnery & Xiao, 2005).

Contudo, os códigos limitados de 6-bit foram também sentidos na década de 1950 levando, assim, à criação de um novo código de 7-bit denominado por ASCII (código padrão americano para troca de informação). Deste houve algumas versões com as seguintes diferenças:

Tabela 3 - Formatos ASCII (McEnery & Xiao, 2005)

Formato	**Comentário**
ASCII-1963	Primeira versão. Não inclui letras minúsculas, embora tenha posições disponíveis para tal.
ASCII-1967	Contém 96 caracteres de impressão, incluindo letras minúsculas, e 32 caracteres de controlo.

[1] O código Morse é um sistema de representação de letras, números e sinais de pontuação através de sinais elétricos longos e curtos, o qual foi inventado pelo americano Samuel Finley Breese Morse (1791-1872) em 1835 (McEnery & Xiao, 2005).

O ASCII foi adotado por quase todos os fabricantes de computadores e posteriormente tornou-se num padrão internacional (ISO 646) pela *International Standard Organization* (ISO) em 1972 (McEnery & Xiao, 2005).

O ASCII de 7-bit é suficiente para codificar caracteres, por exemplo, em Inglês, mas com a crescente necessidade de troca de informação entre múltiplas línguas, incluindo caracteres acentuados (por exemplo, Línguas Europeias) esta codificação rapidamente se tornou inadequada. Assim sendo, o ASCII de 7-bit foi estendido para 8-bit o que já permitiu o suporte a linguagens Europeias. Com este foi formulado um novo padrão (ISO 2022) que já serviu, também, de base para os códigos de 16-bit usados nos países do Este Asiático, tais como, China, Japão e Korea (McEnery & Xiao, 2005).

Com a herança de codificações, quase que cada linguagem tem o seu próprio conjunto de caracteres o que, por um lado, é eficiente para suportar uma língua em específico, mas por outro constitui uma "Torre de Babel", a qual perturba a comunicação internacional. Como tal, têm sido vários os esforços para unificar estes códigos de caracteres incompatíveis entre si, com o objetivo de criar um padrão unificado e global. Estes esforços tiveram início na primeira metade dos anos 80, que curiosamente coincidiu com o início da Internet, e foram prosseguidos por três grupos independentes dos Estados Unidos, Europa e Japão. De entre muitas variantes padronizadas (ISO), surgiu finalmente o Unicode (código de unificação), que foi inicialmente desenhado como um código de tamanho fixo, usando 16 *bits* (2 *bytes*) por cada caracter e disponibilizando espaço para 65.536 caracteres. O Unicode foi criado com o objetivo de ser utilizável em todas as plataformas independentemente do fabricante, fornecedor, *software* ou local. Adicionalmente, para facilitar a troca de informação entre sistemas de diferentes países, também permite que um único documento contenha textos de diferentes sistemas de escrita, o que era quase impossível com os códigos nativos. Assim sendo, o Unicode tornou possível a criação de verdadeiros documentos multilíngue (McEnery & Xiao, 2005).

2.3.1 Unicode

"Internationalization is completely possible without Unicode… but internationalization is much easier with Unicode" (Gillam, 1999)

Muitos dos idiomas asiáticos não usam um sistema alfabético, pois estes ilustram palavras com caracteres pictóricos (por exemplo, glifos e ideogramas). Daqui podem surgir problemas, pois algumas aplicações apenas suportam idiomas com caracteres de byte único e os idiomas asiáticos precisam de caracteres de, pelo menos, dois bytes ou até mais. Por esta razão, os produtos de *software*, que são traduzidos para o japonês, chinês ou coreano, devem apoiar o sistema de byte duplo ou devem ser escritos em Unicode (Gross, 2006).

O Unicode é um padrão internacional, independente da plataforma, que define um código digital para cada caractere, glifo e ideograma de todas as línguas conhecidas em todo o mundo. Este já é integrado em todas as linguagens de programação modernas, tais como Java e C#, através de bibliotecas que estão disponíveis gratuitamente (Gross, 2006).

Com o Unicode os textos multilíngues são codificados de uma forma padronizada, de modo a que a informação seja trocada internacionalmente. Neste sentido, o Unicode é bastante importante, pois permite a criação de um sistema de *software* que funciona no mundo inteiro, sendo este o princípio básico para o *software* global (Gross, 2006).

Combinando todas as letras e caracteres de todas as linguagens existentes no mundo, existem muitos mais caracteres que um conjunto de caracteres de 8-bit pode suportar, para além de que novos caracteres podem surgir ao longo do tempo, como aconteceu por exemplo com o símbolo do Euro (€). Daqui advêm vários temas, como por exemplo, ficheiros de texto simples não conseguirem guardar caracteres de diferentes conjuntos de caracteres. Este tipo de problemas podem ser resolvidos adotando um sistema usável em todo o mundo para codificar conjuntos de caracteres, como é o caso do Unicode (Gross, 2006).

O Unicode não define apenas a identidade de cada caracter e o seu valor numérico, mas também formula como este valor é representado em *bits* quando o caracter é guardado num ficheiro ou transferido através da Internet. As fórmulas deste tipo são referidas como Formatos de Transformação Unicode (UTFs). O padrão Unicode disponibilizou por ordem cronológica 3 UTFs – UTF-16, UTF-8 e UTF-32. Todos codificam o mesmo repositório de caracteres e podem ser transformados de um formato para outro sem perda de informação (McEnery & Xiao, 2005).

Na Tabela 4 são apresentadas as principais características entre cada um dos formatos UTF:

Tabela 4 - Formatos Unicode UTF (McEnery & Xiao, 2005)

Formato	**Características**
UTF-16	• Não é compatível com ASCII. • Usa pelo menos 2 *bytes* por cada caracter independentemente da sua posição no plano básico multilíngue. • Codifica os caracteres mais incomuns (por exemplo, os chineses) através de pares de posições da cadeia de caracteres. • Duplica sempre o tamanho de um ficheiro com caracteres de byte único (por exemplo, Inglês) comparativamente ao UTF-8.
UTF-8	• É compatível com ASCII. • Evita a necessidade de reescrever *software* anterior. • Usa pelo menos 1 *byte* por cada caracter, 2 *bytes* por cada caracter acentuado e 3 *bytes* por cada caracter Chinês. • Os ficheiros tendem a ser mais pequenos que os codificados em UTF-16 ou UTF-32.
UTF-32	• Não é compatível com ASCII. • Consegue codificar cada um dos caracteres, até os mais incomuns, com apenas uma posição da cadeia de caracteres. • Consome muita memória e espaço em disco.

O UTF-8 é considerado o formato universal e a forma mais popular de troca de informação em Unicode, uma vez que é compatível com sistemas já existentes (McEnery & Xiao, 2005). No entanto, ainda há um grande problema com o qual os programadores devem estar cientes: A combinação de codificações usadas na apresentação/entrada de informação e no armazenamento da mesma, pode ser perigosa. Por exemplo, quando o *front-end* de uma aplicação (camada de apresentação) usa ISO8859-1 para a exibir a informação ao utilizador e a base de dados usa UFT-8, se houver a necessidade deste *software* suportar caracteres asiáticos, a mesma base de dados pode continuar a ser utilizada sem problemas, por outro lado, a codificação do *front-end* terá que ser modificada, caso contrário os caracteres asiáticos não serão apresentados corretamente ao utilizador (Gross, 2006).

2.4 Soluções para internacionalização de *software*

Na Figura 2 é apresentado um exemplo de como deve ser feito o desenvolvimento de um *software* internacionalizado. Primeiro, as *strings* são extraídas do código fonte, ficando apenas o código necessário para obter as mesmas em tempo de execução. Estas

strings são armazenadas em *resource bundles*, atribuindo a cada uma delas uma chave única. No código fonte permanece apenas as chamadas destas *strings* através da chave única atribuída anteriormente. Após este passo, não existirá mais informações relativas ao idioma espalhadas pelo código fonte da aplicação. O próximo passo é fazer a localização e a tradução das *strings* resultando, assim, num *software* internacional (Cordioli, 2006).

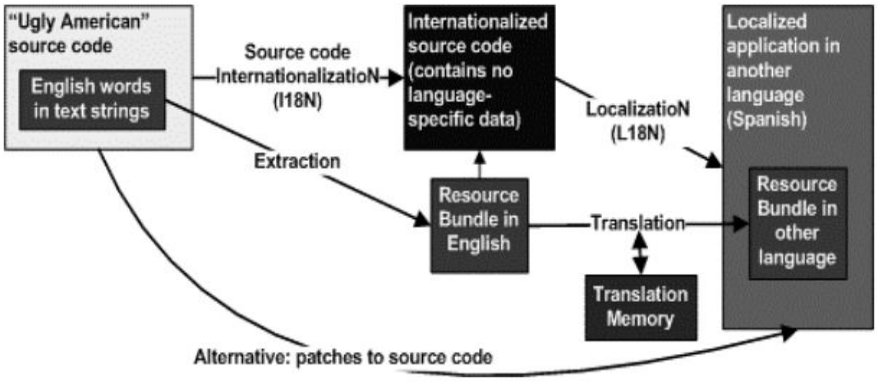


Figura 2 - Diagrama de desenvolvimento de software internacionalizado (Cordioli, 2006).

Desta forma, a internacionalização de *software* geralmente, mais particularidade menos particularidade, anda muito em torno da utilização de pacotes de recursos. Estes pacotes podem ser usados, não só para localizar elementos de texto, mas também, convenções de formatação, imagens, ficheiros e áudio, entre outros (Deitsch & Czarnecki, 2001).

Hoje em dia já existem várias *frameworks* I18n para desenvolvimento de aplicações internacionalizadas. A utilização de pacotes de recursos permite uma mais fácil edição e evita a necessidade de recompilar o código-fonte para cada localização (Batista, 2014). Desta forma, uma aplicação devidamente internacionalizada permite uma fácil localização da mesma. Segundo Gross, 2006, e Batista, 2014, a estrutura alto nível dos processos e componentes envolvidos no processo L10n de *software* é a seguinte:

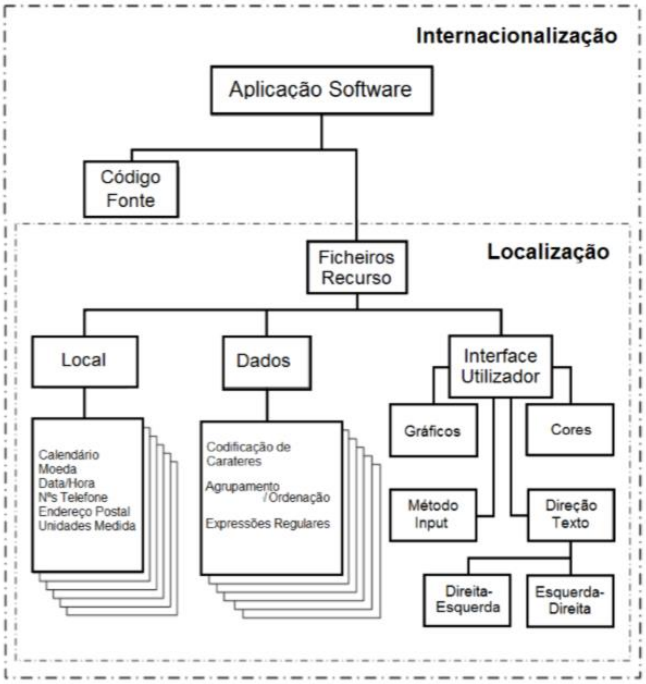


Figura 3 - Divisão de componentes (Batista, 2014; Gross, 2006)

No seguimento da Figura 3, entende-se como "Local" o conjunto de definições regionais associadas a um país ou região. "Dados", a forma como são tratados os caracteres, nomeadamente nas ordenações, agrupamento e transliterações. Por fim, a estrutura "Interface Utilizador" compreende os elementos que constituem a interface gráfica disponibilizada ao utilizador (exemplos: texto, cores, gráficos). O importante é que nenhum destes componentes sejam codificados no código fonte do programa. Todas estas definições devem ser mantidas fora do código-fonte, mas acessível ao *software* (Batista, 2014).

São várias as tecnologias que usam a lógica de pacotes de recursos, habitualmente conhecidos por *resource bundles*. Estes pacotes contêm elementos constituintes da interface do utilizador, tais como, legendas, mensagens, ícones, imagens ou até mesmo *layouts* de janelas inteiras. Tal como exemplificado na Figura 4, o código da interface do utilizador baseia-se nos itens armazenados nestes pacotes de recursos para produzir a respetiva interface (Gillam, 1999).

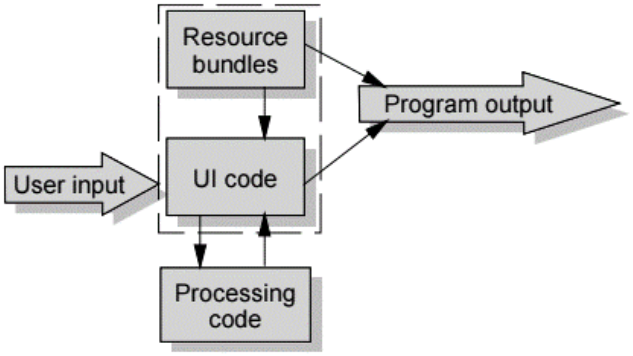


Figura 4 – Lógica dos Resource Bundles (Gillam, 1999).

Dando o exemplo de uma aplicação com arquitetura de três camadas (dados, negócio e apresentação), de acordo com a Figura 5, as *frameworks* de internacionalização normalmente são usadas nas camadas de apresentação e negócio. No que toca aos dados, estes são codificados e transferidos entre as três camadas em formato Unicode, pois desta forma garante-se o suporte de maior parte dos idiomas (Xoriant, 2002).

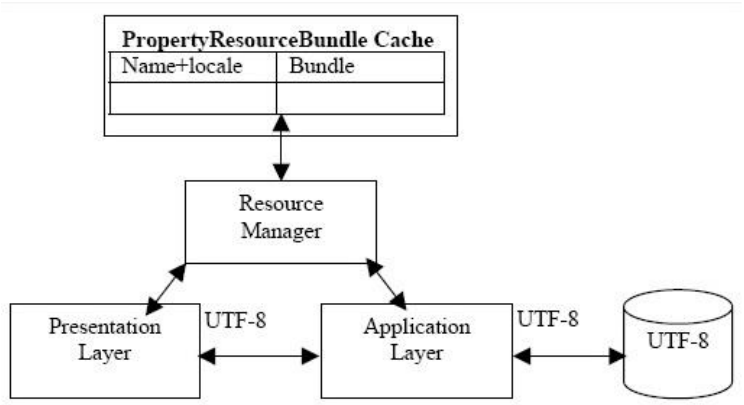


Figura 5 - Arquitetura de aplicação em três camadas (Xoriant, 2002).

Os pacotes de recursos constituem uma componente muito importante na internacionalização de uma aplicação, porque separa o código da informação que precisa ser localizada. Estes pacotes podem ser colocados numa diretoria única, mas são organizados por uma hierarquia lógica baseada nos idiomas associados a cada um deles, estando na base desta hierarquia o pacote de recursos raiz. Posteriormente, quando é necessário localizar a aplicação para um *locale* específico é criado um novo pacote de recursos que traduz toda a informação contida no pacote raiz para o novo *locale* (Wave, 2011).

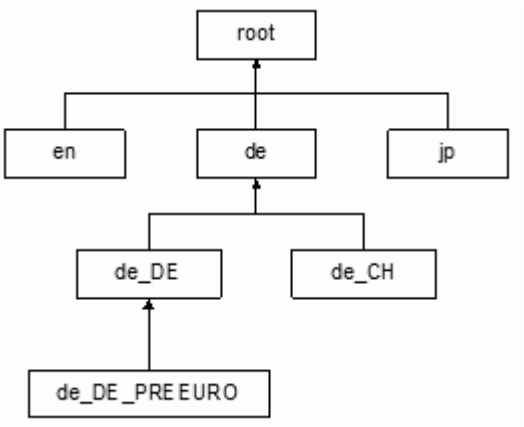


Figura 6 - Hierarquia de resource bundles (Wave, 2011).

A Figura 6 apresenta a estrutura hierárquica dos pacotes de recursos, a qual é composta por quatro níveis: o primeiro é o pacote raiz, o segundo identifica a linguagem, o terceiro o país e o quarto a variante. Com esta hierarquia não é necessário que o valor dos recursos se repitam de pais para filhos, isto é, o mecanismo de regressão garante que um pacote de recursos filho precisa apenas de conter os recursos cujos valores diferem do pacote ancestral. Caso um recurso não seja encontrado no pacote filho este será pesquisado no pacote de nível superior e assim sucessivamente. Caso esta pesquisa chegue ao pacote raiz sem obter o recurso este direcionará a pesquisa para o próximo pacote (Wave, 2011).

Em algumas arquiteturas de internacionalização a abordagem segue ainda pela existência de três grandes componentes, sendo estas a de formatação, comparação e deteção de limites. A Figura 7 representa graficamente esta abordagem.

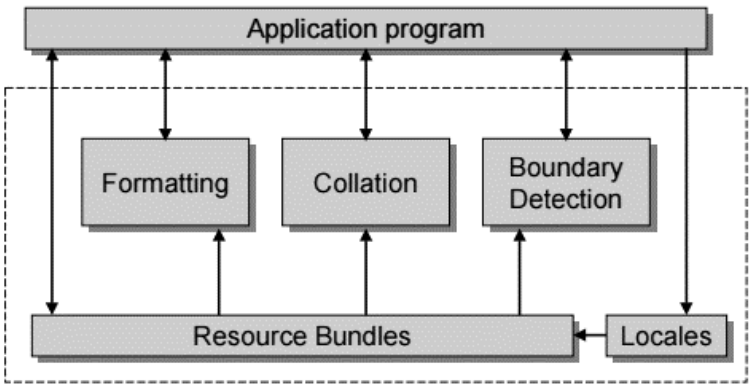


Figura 7 - Arquitetura I18n (Gillam, 1999).

Todas estas componentes dependem dos *resource bundles* pois usam um modelo orientado a dados, isto é, o comportamento de cada uma delas deriva de algum tipo de

descrição textual fornecido pelo chamador ou obtida a partir do próprio *resource bundle*. Esta é uma característica central deste tipo de arquiteturas de internacionalização que permite mudanças de comportamento sem alterar o código (Gillam, 1999).

Colocar elementos de texto, entre outros, em pacotes de recursos já é uma grande mais-valia para tornar a aplicação fácil de traduzir. No entanto, há muito mais que pode ser feito para dinamizar a internacionalização e é aqui que os três componentes referidos na Figura 7 se tornam importantes (Gillam, 1999).

Começando pela formatação de dados para apresentação ao utilizador, é nesta que muitas convenções culturais são aplicadas, como por exemplo, números, datas, horários e mensagens podem requerer formatação antes de serem exibidos (Oracle, 2014). Passando a dar apenas um exemplo, muito simples e muito útil, da utilização deste componente é a formatação de uma mensagem que contém valores dinâmicos, como por exemplo:

- A pesquisa encontrou 23 ficheiros que contêm "Olá" no disco "MeuDisco".

Partindo do princípio que os elementos de texto sublinhados na mensagem acima podem ser variáveis, poderia se pensar numa solução para construção da mesma como a seguinte:

- "A pesquisa encontrou " + hits + " ficheiros que contêm \"" + searchString + "\" no disco \"" + searchRoot + "\"."

Desta forma, para traduzir esta mensagem, teria que se pegar em cada um dos fragmentos da mesma e traduzi-los em separado. Esta solução é um pouco caótica tanto ao nível da internacionalização (código fonte) como no processo de tradução de mensagens, uma vez que a estrutura das frases difere de língua para língua, correndo, até, o risco de diferir na própria ordem dos valores dinâmicos. No entanto, o utilizador quando visualiza a mensagem final tem noção que esta é uma frase única, na qual existem partes dinâmicas que são preenchidas com valores, mas nunca invalidando o facto de esta mensagem ser uma única unidade (Gillam, 1999). É neste sentido que a componente de formatação é útil, uma vez que permite fazer a formatação da frase exatamente como é percecionado pelo utilizador, isto é, a frase completa corresponde a uma única unidade/tradução contendo partes dinâmicas contidas na mesma. Assim, podemos ter a seguinte frase no pacote de recursos:

- A pesquisa encontrou {0} ficheiros que contêm "{1}" no disco "{2}"

Com auxílio da componente de formatação, associando um *array* de valores, os elementos sublinhados na frase acima são substituídos pelos valores do respetivo *array*. A frase em si a ser usada pode ser obtida, já devidamente traduzida, pelo *resource bundle*. Por exemplo, a estrutura da mesma frase em alemão seria (Gillam, 1999):

- Es gibt {0} Dateien auf Platte „{2}", die „{1}" enthalten.

Relativamente ao componente de comparação, a sua principal característica é a capacidade de comparar *strings* sensíveis ao idioma, a qual costuma ser bastante útil para construir rotinas de pesquisa ou ordenação cujas regras variam consoante o *locale* em uso (Oracle, 2014). Sendo que para efetuar pesquisas e ordenações de texto é inevitável a comparação do mesmo, seguem alguns exemplos de temas que podem advir destas comparações, caso não se use uma componente de comparação sensível ao idioma (Gillam, 1999):

- Diferentes definições de "letra"
    - Em inglês, "a" ≈ "ä" e "v" ≠ "w".
    - Em sueco, "a" ≠ "ä" e "v" ≈ "w".
    - Em espanhol, "ch" e "ll" são consideradas uma única letra e não um par de letras.
- Expansão de sequência de caracteres
    - Em alemão, "ä" ≈ "æ" e "ß" ≈ "ss".
- Caracteres ignorados
    - Várias línguas têm implícito caracteres que são ignoráveis em pesquisas e ordenações, como por exemplo, em inglês "e-mail" e "email" são exatamente a mesma palavra.
- Níveis de equivalência para pesquisa
    - Diferenças primárias (letras): "resume" vs "repeat".
    - Diferenças secundárias (diacríticos): "résumé" vs "resume".
    - Diferenças terciárias (capitalização): "RESUME" vs "resume".
    - Diferenças quaternárias (identidade): Quando não existe diferenças terciárias, mas existe ainda diferenças ao nível dos códigos hexadecimais (palavra iguais, mas com caracteres não pertencentes à linguagem) e,

assim, serão ordenados consoante o seu valor hexadecimal. Quanto às pesquisas, a definição de uma palavra também pode variar consoante a idioma.

Relativamente ao componente de deteção de limites, localizar os limites entre palavras não é assim tão simples quanto parece, pois, para além da definição de uma palavra depender do idioma, esta depende, também, do que se pretende fazer. Por exemplo, a melhor posição para aplicar uma quebra de linha num texto em inglês, normalmente, será entre um espaço em branco e um caractere. Em tailandês ou chinês, por exemplo, esta regra já não se aplica, pois não são usados espaços em branco entre palavras. No caso do chinês, apesar de não usar espaços em branco entre as palavras, pode-se aplicar a quebra de linha em praticamente qualquer posição, salvo se existirem sinais de pontuação que devem ser sempre mantidos antes ou depois do caractere (Gillam, 1999).

Figura 8 - Deteção de limites das palavras (Gillam, 1999).

Tendo isto em conta, torna-se importante o componente de deteção de limites com base no idioma, pois criar um algoritmo simples para o efeito corre-se o risco de não obter os resultados corretos em todos os idiomas (Gillam, 1999).

2.4.1 Internacionalização em bases de dados

É importante fazer a distinção adequada dos valores dos dados que devem permanecer inalterados quando se muda o idioma e o formato no qual os dados são representados. O principal foco na conceção de uma base de dados, que permita o armazenamento de conteúdo localizado, é identificar quais os valores que necessitam de ser armazenados de uma forma dependente da cultura e quais os que podem ser armazenados com uma

representação uniforme, podendo ser posteriormente convertidos por lógica da aplicação. A representação de dados numa base de dados deve ser o mais independente possível da localidade, para que a conversão da representação original seja realizada sem perda de informação (Gut, Miclea, Enyedi, Abrudean, & Hoka, 2006).

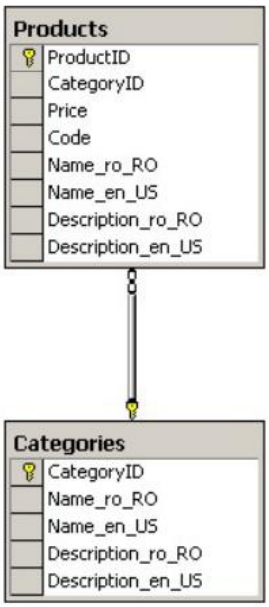
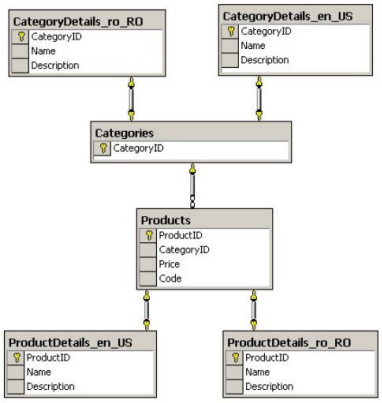
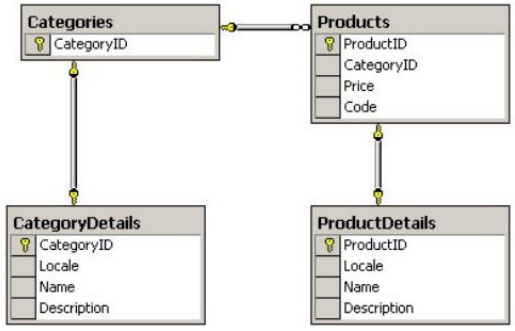
A aproximação recomendada para persistir cadeias de caracteres de diferentes codificações numa base de dados é usar uma codificação universal (UTF-8) como atributo da própria base de dados e converter os formatos originais para o formato da base de dados antes de armazenar os dados na mesma. Podendo estes ser convertidos e apresentados posteriormente por outra codificação (Gut et al., 2006).

Relativamente a valores de moedas, apenas as regras de negócio da organização em questão devem ditar a forma como estes são guardados na base de dados em cada momento. Adicionalmente, quando ocorre uma conversão entre moedas, seja por que motivo for, a taxa de câmbio desta conversão, à data, deve ser também guardada por questões de auditoria. No caso das datas, é habitualmente usada a coordenada de tempo universal (UTC) para representação das mesmas, com indicação do respetivo fuso horário caso seja necessário, pois a partir desta é sempre possível determinar a hora local de qualquer ponto geográfico. Desta forma, nenhuma informação é perdida nestas conversões. Tal como acontece com a moeda, esta determinação depende das regras de negócio de cada organização (Gut et al., 2006).

Num modelo de dados existem, habitualmente, entidades com atributos dependentes do idioma, como por exemplo, legendas, textos descritivos e/ou imagens. Neste sentido, uma das formas mais comuns de internacionalizar uma base de dados é aplicar relações de um-para-muitos a estas entidades. Assim, para um mesmo item de uma determinada entidade é possível inserir nome e atributos de descrição para cada idioma suportado. Na tabela abaixo apresentam-se algumas aproximações a esta solução, tendo como base os seguintes pressupostos:

- Trata-se de uma base de dados para suportar informação sobre produtos e categorias (“Products” e “Categories”) desses mesmos produtos.
- Como é suposto armazenar produtos em categorias e uma categoria pode conter vários produtos, cada produto terá uma referência para a categoria a que pertence (relação um-para-muitos).

Tabela 5 – Aproximações I18n em bases de dados (Gut et al., 2006)

Modelo	**Observação**
	Neste modelo, os campos *"Name"* e *"Description"* são os mais prováveis de requererem internacionalização. Assim sendo, adicionou-se um novo atributo por cada um deles para os vários idiomas suportados (neste caso, ro_RO e en_US). A maior desvantagem nesta aproximação é, no suporte a um novo idioma, exigir alterações ao nível do código, na logica de acesso aos dados, bem como alterações ao nível da estrutura da própria base de dados. Outra desvantagem é que todas as colunas, para além das colunas do idioma em uso, são inúteis e, em casos que se pretende suportar mais que um ou dois idiomas, a manipulação e consulta às respetivas tabelas torna-se bastante ineficiente.
	Uma opção é manter as tabelas base (*"Products"* e *"Categories"*), mas colocar a informação localizada em entidades separadas por cada idioma suportado. Esta opção é melhor que a anterior, mas ainda assim requer sempre alterações ao nível da estrutura da base de dados por cada novo idioma (criação de novas tabelas descritivas).
	Uma variante do modelo anterior é manter, de igual forma, as tabelas base (*"Products"* e *"Categories"*), no entanto, a informação localizada passa estar em tabelas de detalhes genéricas, sendo o *locale* definido ao nível da coluna e não da tabela em si. De entre as apresentadas, esta é a aproximação mais recomendada, pois torna o idioma como parte integrante da entidade que representa a informação localizada.

Modelo	Observação
	Usando esta abordagem, não é necessário nenhuma lógica de localização no código da aplicação, sendo esta transportada para o lado do servidor de base de dados, passando apenas a indicação do idioma como parâmetro de consulta ou de entrada dos procedimentos armazenados.

Projetar uma base de dados para uma aplicação internacional exige uma enorme capacidade de desenho e visão do panorama global da mesma, mas fornece, também, uma grande flexibilidade para suportar conteúdo localizado mais tarde. Desta forma, é importante ter em conta a entrada e saída de dados, apresentação e codificação dos mesmos, no processo de internacionalização (Gut et al., 2006).

2.5 Acesso a dados em memória

A maneira mais rápida de aceder aos dados é já dispor deles em memória no momento de uso. A implicação da adoção desta proposição é fazer uma pré-carga de toda a informação necessária para evitar o impacto do acesso demasiado à base de dados.

Por exemplo, se os dados de um determinado item estão distribuídos por algumas tabelas e se este item for utilizado neste ambiente, então é possível carregar toda esta informação no momento de inicialização do sistema e mantê-la em memória. Isto é particularmente útil e viável, se esta informação for imutável (Kozovits, Melo, & Feijó, 2003).

3 Modelo conceptual

Tal como referido anteriormente, esta parte será constituída pela elaboração de um desenho de visão a par do estudo da arquitetura da base de dados em desenvolvimento. A revisão de literatura terá alguma influência na elaboração da primeira tarefa desta parte da dissertação. Já a segunda tarefa requer também alguma investigação técnica referente à respetiva base de dados, no sentido de perceber a sua constituição e características de implementação.

3.1 Desenho de visão

Com base na revisão de literatura chegou-se a alguns pressupostos importantes a ter em conta na elaboração deste modelo de internacionalização, sendo estes os seguintes:

- Separar o conteúdo localizado do código fonte da aplicação.

  É importante separar todo o conteúdo localizável do código fonte da aplicação para que este possa ser acedido e traduzido por pessoas sem qualquer conhecimento de programação e/ou bases de dados. Esta prática salvaguarda sempre o correto funcionamento da aplicação uma vez que não é necessário disponibilizar o código fonte da mesma a pessoas sem este tipo de conhecimento. Desta forma, evitam-se possíveis erros na aplicação provenientes de alterações erróneas por parte de profissionais não habilitados para a programação.

- Equipas de desenvolvimento e tradução devem trabalhar a par.

  Um pouco relacionado com o ponto anterior, os elementos das equipas de tradução não têm conhecimentos de bases de dados, mas ainda assim têm que trabalhar em conjunto com as equipas de desenvolvimento. Daqui advém a necessidade de colocar todo o conteúdo localizável em ficheiros externos para que sejam facilmente acedido e alterados pelas equipas de tradução sempre que necessário. Normalmente a relação é de um ficheiro por idioma.

- Consultas mais rápidas com auxílio de *cache*.

O tempo de acesso à *cache* é substancialmente menor que o acesso a tabelas ou ficheiros. Assim sendo, é boa política carregar a informação necessária à cabeça, assim que o sistema inicia, principalmente se esta informação for imutável a curto prazo.

- Evitar períodos de indisponibilidade da base de dados.

  Em caso de manutenção do conteúdo localizável é suposto não ser necessário indisponibilizar a base de dados, pois para uma base de dados de produção é importante que a mesma esteja disponível o máximo de tempo possível.

Posto isto, pretende-se obter uma função para internacionalização que possa ser invocada tanto numa *query* como num procedimento ou função e que a mesma esteja contida no próprio motor da base de dados. Por outro lado, pretende-se que os ficheiros de legendas estejam facilmente acessíveis por tradutores que não têm qualquer conhecimento em manipulação de bases de dados.

Neste sentido, uma possível solução é criar uma nova palavra SQL não reservada que, estando associada a um método de manipulação I18n, faz a consulta das legendas que o programador deseja, através de códigos passados como parâmetro de entrada na respetiva função. Cada um destes códigos está associado a uma única legenda por cada língua e cada língua corresponde a um ficheiro distinto de legendas.

Sempre que se adicionam legendas novas, para que não seja necessário reiniciar a base de dados esta deve ter a capacidade de consultar os ficheiros de legendas externos sempre que necessário. Por outro lado, para não tornar este processo bastante pesado no que diz respeito à performance, o manipulador I18n da base de dados deve, no momento da consulta da legenda, consultar primeiro as legendas que se encontram carregadas na *cache* da base de dados e caso não encontre deverá, então, consultar os ficheiros de legendas externos. Caso encontre a legenda nos ficheiros externos deverá devolvê-la e carregá-la na *cache*, para uma próxima que seja solicitada, esta já se encontrar devidamente carregada. Por outro lado, no caso de se editar nos ficheiros legendas já previamente carregadas na *cache*, esta não vai ser modificada na *cache* porque já lá existe. Desta forma, deverá existir a possibilidade de remover a respetiva legenda da cache para que esta volte a ser carregada já devidamente editada. Por fim, sempre que a base de dados é iniciada é efetuado o carregamento de todas as legendas para a *cache*.

Na Figura 9 é resumidamente ilustrado o modelo do processo descrito acima.

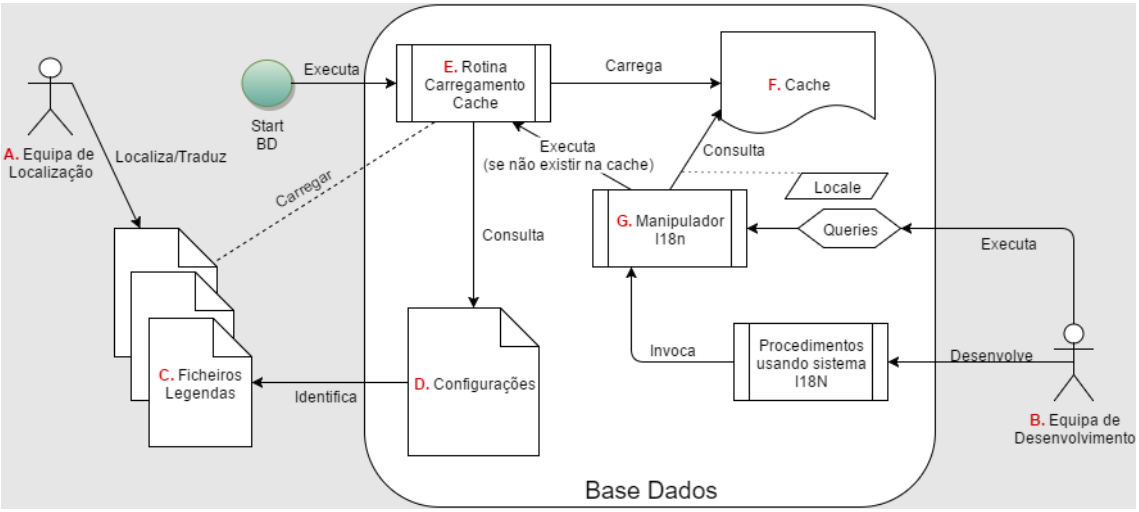


Figura 9 - Desenho visão do modelo I18n em base de dados.

No ponto A tem-se a equipa de tradução que irá traduzir o conteúdo em C, que diz respeito aos ficheiros externos. Por outro lado, no ponto B, tem-se a equipa de desenvolvimento que produz o *software* e, por sua vez, cria procedimentos, funções e executa *queries* na base de dados. Quando a base de dados inicia a rotina no ponto E é automaticamente executada carregando, assim, o conteúdo dos ficheiros em C na *cache*, representada em F, com auxílio das configurações do ponto D. Estas configurações são propriedades de base de dados criadas para definir a localização dos ficheiros externos no sistema de ficheiros e o idioma do ficheiro padrão. Por fim, sempre que é feito um pedido de uma legenda por parte dos desenvolvimentos da equipa em B, o manipulador I18n, representado no ponto G, efetua a consulta na *cache* (F) e, caso não obtenha resultado, efetua a consulta nos ficheiros externos (C) com auxílio da rotina de carregamento (E).

Conforme ilustrado na Figura 10, este modelo pode dividir-se em quatro principais fases de implementação. A primeira (azul) onde se cria a nova palavra SQL não reservada e se começa a desenvolver o manipulador I18n. A segunda (amarelo) onde se efetua a consulta dos ficheiros externos. A terceira (vermelho) onde é feito o carregamento inicial da *cache*. E, por fim, a quarta (verde) onde se dá a consulta da *cache* ou dos ficheiros por parte do manipulador.

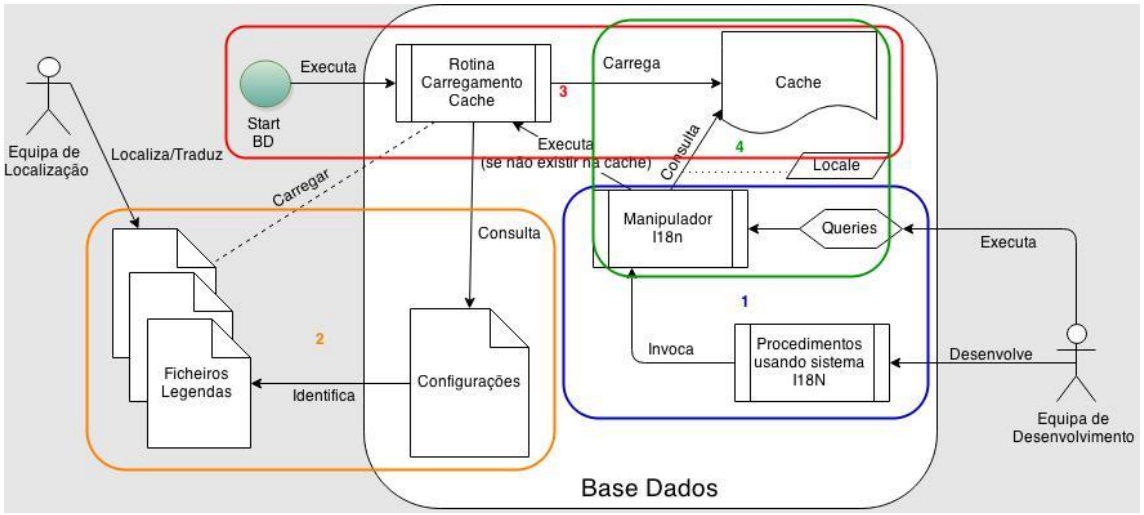


Figura 10 - Fases de implementação do modelo I18n.

4 Protótipo

4.1 Investigação tecnológica

Foi feita investigação sobre algumas tecnologias que já têm como parte integrante a internacionalização através de ficheiros de recursos, tendo sido as tecnologias Java e Zend Framework investigadas em detalhe e documentadas nos pontos 4.1.1 e 4.1.2 desta dissertação. No âmbito destas tecnologias foram investigadas as respetivas arquiteturas e utilização no que toca à internacionalização de *software*. A escolha das mesmas deve ao facto destas serem das mais populares e conterem funcionalidades muito semelhantes ao que se pretende desenvolver.

Posteriormente foram investigados alguns SGBDs de forma a conhecer as suas arquiteturas com o objetivo de aplicar um módulo de internacionalização semelhante ao anterior, por ficheiros de recursos. Dos motores de bases de dados investigados o escolhido para a concretização do modelo de internacionalização foi o HSQLDB.

4.1.1 JAVA - Resource Bundle

Um *resource bundle* em Java é, conceptualmente, um conjunto de classes relacionadas que derivam da classe "ResourceBundle". Cada um destas subclasses tem o mesmo nome base, mais uma componente adicional que identifica o seu *locale*. Por exemplo, supondo que o *resource bundle* é nomeado de "MyResources", a primeira classe que será a padrão terá exatamente o mesmo nome. Posteriormente, pode-se criar tantas classes relacionadas quantas forem necessárias, mas desta vez com a identificação do *locale* junto ao nome base. O objetivo é que cada classe relacionada contenha os mesmos itens, mas respetivamente traduzidos para o idioma que cada subclasse representa (Xoriant, 2002). Esta estrutura será detalhada mais à frente neste capítulo.

A classe "ResourceBundle" tem mais duas subclasses chamadas "PropertyResourceBundle" e "ListResourceBundle" às quais não se interage diretamente, pois todas as interações com estas ocorrem através da classe principal "ResourceBundle" (Jenkov, 2014).

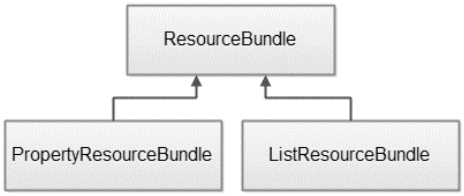


Figura 11 - Classes de suporte à "ResourceBundle" (Jenkov, 2014).

A subclasse "PropertyResourceBundle" é responsável por gerir os recursos por idioma, usando um conjunto de *strings* localizadas que estão armazenadas em ficheiros de propriedades pedrão do Java. Com esta subclasse não é necessário ter mais nenhuma subclasse de *resource bundle*, pois desta forma a classe principal "ResourceBundle" automaticamente acede ao ficheiro de propriedades apropriado e cria o "PropertyResourceBundle" referente ao mesmo (Xoriant, 2002).

É possível carregar estas propriedades através da classe "ResourceBundle", usando o método "getBundle", tal como apresentado na Figura 12.

```
Locale locale = new Locale("en", "US");
ResourceBundle labels = ResourceBundle.getBundle("i18n.MyBundle",
                                                 locale);

System.out.println(labels.getString("label1"));
```

Figura 12 - Carregar e usar propriedades resource bundle Java (Jenkov, 2014).

Tendo um ficheiro de propriedades padrão do Java, chamado "MyBundle.properties", dentro do *package* "i18n" (nome base do *resource bundle*, "i18n.MyBundle"), este será carregado ao ser chamado o método "getBundle", indicando o nome base e o *locale* pretendido. Após o carregamento, pode-se então aceder às suas propriedades através do método "getString" indicando a chave da propriedade que se pretende obter. No exemplo da Figura 12 o conteúdo do ficheiro de propriedades seria conjuntos de chave valor, à semelhança dos seguintes (Jenkov, 2014):

- label1 = Label 1 is done!
- label2 = Label 2 is through!

Em alternativa aos ficheiros de propriedades, também é possível usar conjuntos de classes para armazenar os recursos. Estas classes derivam da "ListResourceBundle" e com estas é possível usar mais elementos do que apenas elementos de texto. À

semelhança das dos ficheiros de propriedades, são criadas classes com o nome base do *resource bundle* e o idioma como sufixo.

A Figura 13 mostra um exemplo de implementação da classe referente a um pacote de recursos padrão (sem indicação do *locale* no nome base).

```
package i18n;

import java.util.ListResourceBundle;

public class MyClassBundle extends ListResourceBundle {

    @Override
    protected Object[][] getContents() {
        return contents;
    }

    private Object[][] contents = {
            { "price"   , new Double(10.00) },
            { "currency", "EUR" },
    };
}
```

Figura 13 - Classe do resource bundle padrão (Jenkov, 2014).

A Figura 14 mostra um exemplo de implementação da classe referente a um pacote de recursos para o idioma dinamarquês (_da).

```
public class MyClassBundle_da extends ListResourceBundle {

    @Override
    protected Object[][] getContents() {
        return contents;
    }

    private Object[][] contents = {
            { "price"   , new Double(75.00) },
            { "currency", "DKK" },
    };
}
```

Figura 14 - Classe do resource bundle de idioma dinamarquês (Jenkov, 2014).

É de notar que os conteúdos, consistem num *array* bidimensional de chave-valor, em que “price” e “currency” são as chaves e os valores à direita dos mesmos são os valores localizados. Neste exemplo temos preços em moedas diferentes, dependendo do país (Jenkov, 2014).

A obtenção de uma instância "ListResourceBundle" é exatamente igual a uma "PropertyResourceBundle". Com o método "getObejct" obtém-se o valor localizado referente à chave indicada no mesmo.

```
Locale locale = new Locale("de", "DE"); // Padrão
ResourceBundle bundle = ResourceBundle.getBundle("i18n.MyClassBundle",
locale);

System.out.println("price    : " + bundle.getObject("price"));
System.out.println("currency: " + bundle.getObject("currency"));
```

Figura 15 - Utilizar resource bundle padrão (Jenkov, 2014).

O resultado da Figura 15 é o seguinte:

```
price    : 10.0

currency: EUR
```

Inserindo, agora, o *locale* dinamarquês (da_DK) na instância do pacote de recursos a classe a ser consultada será a referente ao idioma dinamarquês (MyClassBundle_da).

```
locale = new Locale("da", "DK");
bundle = ResourceBundle.getBundle("i18n.MyClassBundle", locale);

System.out.println("price    : " + bundle.getObject("price"));
System.out.println("currency: " + bundle.getObject("currency"));
```

Figura 16 - Utilizar resource bundle dinamarquês (Jenkov, 2014).

O resultado da Figura 16 é o seguinte:

```
price    : 75.0

currency: DKK
```

A desvantagem do uso de classes em vez de ficheiros de propriedades é no caso de haver adições de novos elementos ou alterações dos já existentes, as classes necessitam de ser recompiladas e os ficheiros de propriedades não. Assim sendo, normalmente o uso de "PropertyResourceBundle" é mais aconselhado (Xoriant, 2002).

Relativamente à pesquisa do pacote de recurso a ser utilizado, a classe "ResourceBundle" usa um mecanismo específico para encontrar os pacotes de recursos individuais que são solicitados através do método "getBundle". A hierarquia da nomenclatura dos pacotes de recursos forma uma estrutura em árvore. O nome atribuído aos mesmos é a combinação do nome base com a informação do respetivo idioma. O nome base definido corresponde à raiz desta árvore (Deitsch & Czarnecki, 2001). Um exemplo desta estrutura encontra-se na Figura 17, em que o nome base dos pacotes de recursos é "MyResources".

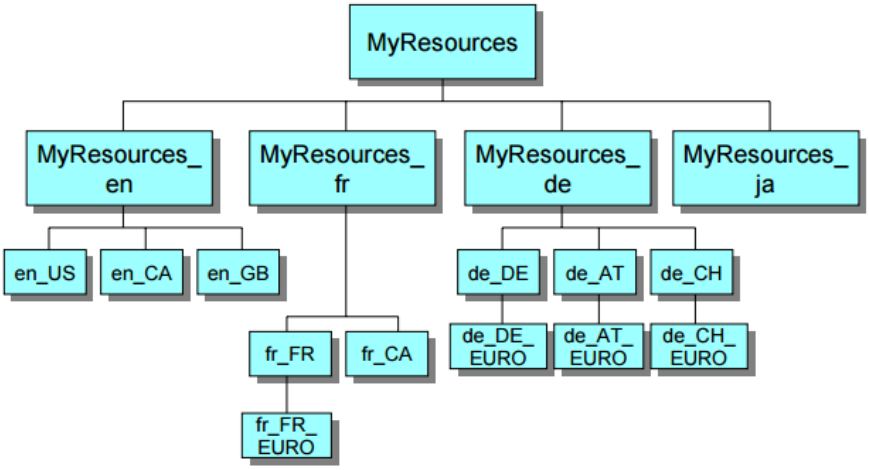


Figura 17 - Hierarquia de Resource Bundles do Java (Gillam, 1999).

É possível diferenciar no segundo nível com o código da linguagem, no terceiro nível com os códigos da linguagem + país e, por fim, no quarto nível também com código da variante (Gillam, 1999).

O método "getBundle" pesquisa por um pacote de recursos específico pela seguinte ordem (Deitsch & Czarnecki, 2001):

1. Pelo *locale* desejado ou solicitado, passado por parâmetro de entrada do respetivo método.
2. Pelo *locale* padrão da JVM.
3. Pelo pacote de recursos padrão, ou com o nome base fornecido.

A pesquisa procede de baixo, num nível mais específico, isto é, mais abaixo na hierarquia de nomenclatura dos pacotes de recursos, para cima, o nível mais genérico. Passando a exemplificar (Deitsch & Czarnecki, 2001):

1. nomaBase + "_" + localeSolicitado.languagem + "_" + localeSolicitado.país + "_" + localeSolicitado.variante
2. nomaBase + "_" + localeSolicitado.languagem + "_" + localeSolicitado.país
3. nomaBase + "_" + localeSolicitado.languagem
4. nomaBase + "_" + localePadrão.languagem + "_" + localePadrão.país + "_" + localePadrão.variante
5. nomaBase + "_" + localePadrão.languagem + "_" + localePadrão.país
6. nomaBase + "_" + localePadrão.languagem
7. nomaBase

Na Figura 18 e Figura 19 é identifica na árvore a ordem de pesquisa dos pacotes de recurso Java.

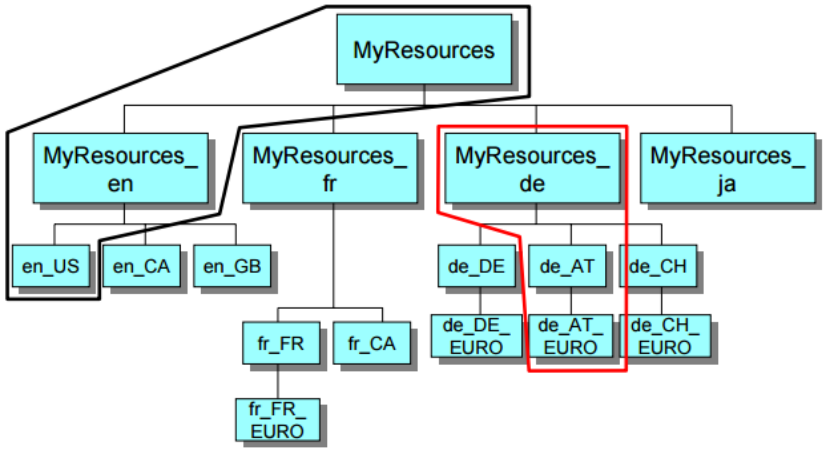


Figura 18 - Ordem de pesquisa de Resource Bundles do Java - Parte 1 (Gillam, 1999).

A linha vermelha identifica o *locale* solicitado (neste caso "de_AT_EURO") e começa a pesquisa de baixo para cima, para o mais generalizado, até encontrar o recurso. Se não for encontrado, é feita a pesquisa da linha preta, também de baixo para cima, referente ao *locale* padrão (neste caso "en_US"), sendo o pacote raiz, o de último recurso. Uma vez que o pacote que pretende (MyResources_de_AT_EURO) existe, então a pesquisa para por aí (Gillam, 1999).

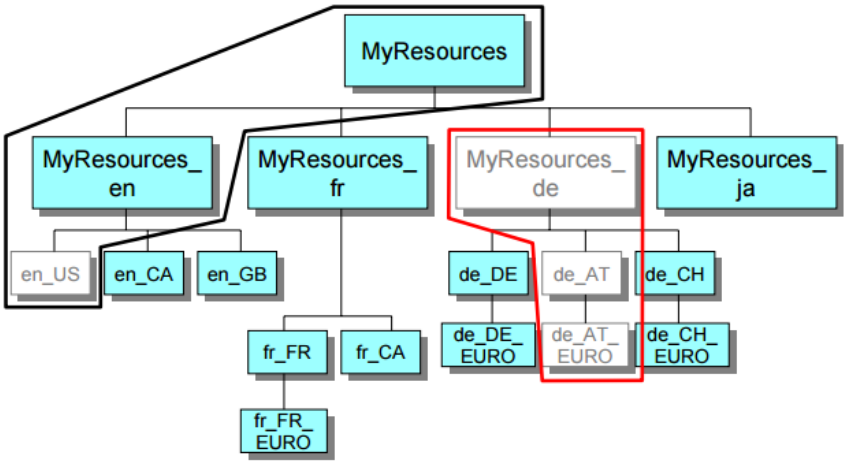


Figura 19 - Ordem de pesquisa de Resource Bundles do Java - Parte 2 (Gillam, 1999).

No caso da Figura 19, não existe nenhuma linguagem alemã e também não há um pacote específico para inglês dos Estados Unidos (*locale* padrão). Assim sendo, se for solicitado Alemão Austríaco a pesquisa cairá sobre o pacote do inglês genérico (Gillam, 1999).

Os pacotes de recursos usados numa aplicação devem estar disponíveis numa diretoria na *classpath* da JVM. Caso contrário, estes não serão encontrados e uma exceção será lançada quando o método "getBundle" for chamado (Deitsch & Czarnecki, 2001).

As facilidades oferecidas pelos pacotes de recursos Java permitem separar as mensagens apresentadas ao utilizador, no *front end*, do código fonte da aplicação, não ficando assim as mesmas codificadas diretamente na aplicação. Por outro lado, não é limitativo que se possa usar apena um único pacote de recursos, pois é possível diferenciar por tipologia os vários recursos por vários pacotes, como por exemplo (Deitsch & Czarnecki, 2001):

- UserExecptionResources
- UserExecptionResources_de
- UserExecptionResources_de_DE
- UserExecptionResources_es
- StatusMessageResources
- StatusMessageResources_es
- StatusMessageResources_es_ES

4.1.2 Zend Framework – Internacionalização

A Zend Framework (ZF) é uma *framework* MVC *open source*, totalmente orientada aos objetos, para desenvolver aplicações *web* e serviços usando PHP5. A estrutura de componentes do ZF é muito peculiar, pois cada componente é implementado com poucas dependências de outros componentes. Esta arquitetura de baixo acoplamento permite aos programadores usar componentes individualmente (Zend, 2015a).

No caso do processo de internacionalização, o mesmo é composto por vários componentes, listados abaixo, tendo cada um destes a sua responsabilidade específica (Mantoan, 2012):

- Zend_Locale: Disponibiliza os *locales* para a localização dentro de outros componentes de internacionalização do Zend Framework;
- Zend_Translate: Traduz os textos dinâmicos, isto é, com possibilidade de substituição de variáveis nos mesmos;
- Zend_Date: Localiza a data e hora;

- Zend_Currency: Localiza as moedas;
- Zend_Locale_Format: Analisa e gera números localizados.
- Zend_Locale_Data: Obtém textos padrão localizados, tais como, nomes de países ou nomes de linguagens.

Dos componentes listados acima, serão detalhados de seguida apenas os componentes Zend_Locale e Zend_Translate, por se tratarem dos mais específicos para a configuração de todo o processo de internacionalização dos demais componentes.

4.1.2.1 Zend_Locale

O principal componente de internacionalização do ZF para identificar o idioma e a região de um determinado utilizador é o Zend_Locale. Esta identificação é feita através de um *locale*, existindo uma lista de combinações possíveis de *locales* a utilizar como referência. Todos os componentes do ZF que suportam internacionalização e localização utilizam este componente para fornecer a normalização e a formatação conforme o idioma e região do respetivo utilizador (Mantoan, 2012).

Abaixo seguem as várias formas possíveis de configurar o Zend_Locale:

- Por um *locale* específico;
  ```
  $forceLocale = new Zend_Locale('pt_PT');
  ```
- Pelo idioma do *browser* do utilizador;
  ```
  $browserLocale = new Zend_Locale(Zend_Locale::BROWSER);
  ```
- Pelo idioma do servidor;
  ```
  $serverLocale = new Zend_Locale(Zend_Locale::ENVIRONMENT);
  ```
- A partir dos parâmetros da *framework*.
  ```
  $frameworkLocale = new Zend_Locale(Zend_Locale::FRAMEWORK);
  ```

Quando não é possível detetar automaticamente o *locale* do utilizador, por exemplo numa linha de comandos, é necessário, assim, configurar um *locale* padrão usando a seguinte instrução: `Zend_Locale::setDefault('pt_BR');` (Mantoan, 2012).

Após o Zend_Locale estar devidamente configurado, é possível utilizá-lo nas classes que suportam L10n, definindo manualmente nos restantes componentes o *locale* a ser usado pelo mesmo (Figura 20) ou definir à cabeça o *locale* para todos os componentes do ZF (Figura 21) (Mantoan, 2012).

```
$enUS = new Zend_Locale('en_US');
$date = new Zend_Date(null, null, $enUS);
echo $date;
```

Figura 20 - Configurar locale para ser utilizado num componente específico (Mantoan, 2012).

```
$ptPT = new Zend_Locale('pt_PT');
Zend_Registry::set('Zend_Locale', $ptPT);
$currency = new Zend_Currency();
echo $currency;
```

Figura 21 - Configurar locale para ser utilizado em todos os componentes do Zend Framework (Mantoan, 2012).

4.1.2.2 Zend_Translate

Após todos os componentes i18n do ZF saberem exatamente o idioma e região do respetivo utilizador é necessário fornecer o conteúdo textual traduzido. Esta função fica a cargo da componente Zend_Translate, a qual dispõe de diversos tipos de fornecedores para as traduções, tais como, *arrays* PHP, arquivos Gettext ou em formato ".CSV" (valores separados por vírgula), entre outros. É nestes ficheiros onde são armazenados todos os textos internacionalizados, existindo um por cada idioma. Na Figura 22 é apresentado o conteúdo de três ficheiros de três idiomas diferentes, "en.php" em inglês, "es.php" em espanhol e "pt.php" em português, os quais estão localizados na pasta "/languages" da aplicação.

```
//en.php
return array(
    'Exemplo' => 'Sample',
    'Internacionalizado' => 'Internationalized',
    'Texto 2' => 'Text 2'
);
// es.php
return array(
    'Exemplo' => 'Ejemplo',
    'Internacionalizado' => 'Internacionalizados',
    'Texto 2' => 'Texto 2'
);
// pt.php
return array(
    'Exemplo' => 'Exemplo',
    'Internacionalizado' => 'Internacionalizado',
    'Texto 2' => 'Texto 2'
);
```

Figura 22 - Conteúdo de ficheiros de traduções (Mantoan, 2012).

No código da Figura 23 começa-se por fazer a instanciação no Zend_Translate com o ficheiro de traduções em inglês. De seguida, são adicionadas ao componente as traduções dos ficheiros português e espanhol.

```
$translate = new Zend_Translate(
    array(
        'adapter' => 'array',
        'content' => '/languages/en.php',
        'locale'  => 'en'
    )
);
$translate->addTranslation(
    array(
        'content' => '/languages/pt.php',
        'locale' => 'pt'
    )
);
$translate->addTranslation(
    array(
        'content' => '/languages/es.php',
        'locale' => 'es'
    )
);
```

Figura 23 - Instânciar Zend_Translate a adicionar idiomas (Mantoan, 2012).

O parâmetro *"adapter"* define qual o tipo de fonte de traduções a ser utilizado no processo de tradução. O *"content"* define a localização onde se encontra a respetiva fonte. Por fim, o *"locale"* define o idioma para a fonte de dados em questão. Posteriormente, pode-se adicionar mais idiomas ao componente utilizando o método "addTranslation" (Mantoan, 2012).

Tendo a configuração completa, é possível utilizar os textos internacionalizados tal como apresentado na Figura 24.

```
// Retorna o texto internacionalizada

$translate->_("Exemplo");

// Imprime o texto internacionalizada

echo $translate->_("Internacionalizado");

// Obtém o texto de acordo com o locale

echo $translate->_("Exemplo", "es");
```

Figura 24 - Obter textos internacionalizados (Mantoan, 2012).

O método "_()" do Zend_Translate recebe como parâmetro de entrada uma chave que identifica o texto internacionalizado, e devolve o mesmo de acordo com o *locale* atual, sendo que esta chave pode ser o texto em si, um índice numérico único, ou uma *string*

única. Adicionalmente, o método tem a opção de aceitar um segundo parâmetro respeitante a um *locale* solicitado especificamente (Mantoan, 2012).

É de notar que, caso exista curingas ("%1\$s", "%2\$s", "%3\$s", …) nas traduções, como por exemplo "Hoje é dia %1\$s", estas poderão ser substituídas por valores dinâmicos da seguinte forma:

- `printf($translate->_("Hoje é dia %1\$s") . "\n", date('d.m.Y'));`

No exemplo apresentado acima a curinga será substituída pela data. Desta forma, uma tradução pode ser feita sem saber o valor exato. Neste caso, a data corresponde ao dia real, mas a cadeia pode ser traduzida sem o conhecimento do dia atual (Zend, 2015b).

É possível, também, definir o *locale* atual da instância de Zend_Translate, com o método "setLocale". No caso de o *locale* definido ser composto por idioma e região (por exemplo, en_GB) e existir um ficheiro somente identificado pelo nome do idioma, o componente procurará pela tradução mais próxima, neste caso seria o "en". Por outro lado, para verificar se uma determinada tradução existe, o método utilizado é o "isAvailable" passando como parâmetro o *locale* que se pretende verificar (Mantoan, 2012).

```
// Mudando o locale, usará a tradução "pt"
$translate->setLocale("pt_PT");
echo  translate->_("Exemplo");

// Verifica se existe um locale
if ($translate->isAvailable("it")) {
    echo $translate->_("Pizza", "it");
}
```

Figura 25 - Alteração do locale da instância Zend_Translate e verificação da existência de traduções de um idioma específico (Mantoan, 2012).

Existem várias funções adicionais ao componente Zend_Translate, tais como, auto detetar os ficheiros de tradução que estão numa determinada diretoria. Neste sentido, ao configurar os parâmetros "*content*" e "*scan*", na instanciação do Zend_Translate, os ficheiros existentes na diretoria "*content*" serão lidos e adicionados automaticamente, sem o auxílio do método "addTranslation" referido anteriormente. Note-se que existem

dois valores possíveis para o "*scan*": "Zend_Translate::LOCALE_DIRECTORY" (o *locale* é detetado ao nível da diretoria) e "Zend_Translate::LOCALE_FILENAME" (o *locale* é detetado ao nível do ficheiro). Para o caso da Figura 22 o que deve ser utilizado é "Zend_Translate::LOCALE_FILENAME" tal como apresenta a Figura 26 (Mantoan, 2012).

```
$translate2 = new Zend_Translate(
    array(
        'adapter' => 'array',
        'content' => 'languages/',
        'scan' => Zend_Translate::LOCALE_FILENAME
    )
);
echo $translate2->_("Exemplo") . "\n";
echo $translate2->_("Exemplo", "es") . "\n";
echo $translate2->_("Exemplo", "en") . "\n";
```

Figura 26 - Auto detetar ficheiros de traduções (Mantoan, 2012).

O Zend_Translate permite, ainda, usar internamente a componente Zend_Cache para manter carregadas as fontes de tradução. Isto é muito útil se forem usadas muitas fontes de tradução ou fontes muito extensas, com por exemplo, ficheiros XML (*Extensible Markup Language*) (Zend, 2015b).

Para usar a *cache* apenas se tem que fornecer um objeto *cache* com "Zend_Translate::setCache()". Este recebe com único parâmetro uma instância de Zend_Cache. A cache pode ser igualmente usada com outros componentes tendo, também, disponíveis os métodos estáticos "getCache()", "hasCache()", "clearCache()" e "removeCache()" (Zend, 2015b).

```
$cache = Zend_Cache::factory('Core',
                             'File',
                             $frontendOptions,
                             $backendOptions);
Zend_Translate::setCache($cache);
$translate = new Zend_Translate(
    array(
        'adapter' => 'gettext',
        'content' => '/path/to/translate.mo',
        'locale'  => 'en'
    )
);

// para limpar a cache algures mais tarde
Zend_Translate::clearCache();
```

Figura 27 - Usar cache com Zend_Translate(Zend, 2015b).

Na figura Figura 27 é criado um objeto Zend_Cache o qual associado ao componente Zend_Translate. Posteriormente é, então, instanciado o objeto Zend_Translate. De outra forma, este não fica armazenado em *cache* (Zend, 2015b).

Quando a cache associada suporta o parâmetro "*tag*" é possível usá-lo para limpar apenas a *cache* de uma instância Zend_Translate em específico. Assim, para apagar um determinada *cache*, basta apenas indicar a "*tag*" como parâmetro de entrada do método "clearCache()" como exemplificado na Figura 28.

```
$cache = Zend_Cache::factory('Core',
                             'File',
                             $frontendOptions,
                             $backendOptions);
Zend_Translate::setCache($cache);
$translate = new Zend_Translate(
    array(
        'adapter' => 'gettext',
        'content' => '/path/to/translate.mo',
        'locale'  => 'en',
        'tag'     => 'MyTag'
    )
);

// algures posteriormente no código
Zend_Translate::clearCache('MyTag');
```

Figura 28 - Limpar uma cache específica (Zend, 2015b).

O Zend Framework dispõe de muitos mais componentes, mas os acima referidos são os mais relevantes no que toca a internacionalização e localização de *software*.

4.1.3 Base de Dados Hypersonic SQL

A bade de dados Hypersonic SQL, mais conhecida por HSQLDB, é *open source*, criada inteiramente em Java e capaz de operar embutida numa aplicação ou como servidor de rede independente. Foi criada em 1998 por um grupo de entusiastas que quiseram explorar a linguagem Java para desenvolver um SGBD que fosse leve e simples (Azenha, 2006). Possui um dialeto SQL considerado mais rico que muitos SGBD's tidos como mais poderosos, como por exemplo, o MySQL. A popularidade da HSQLDB é inegável ao constatar os projetos que a incluem como padrão, como por exemplo, iReport ou o OOo Base, que é o equivalente ao Access do Microsoft Office e faz parte do pacote de ferramentas de escritório OpenOffice. Esta base de dados foi criada com o objetivo de ser leve e com pouca exigência de processador, memória e armazenamento, tendo sido, assim, utilizada com sucesso, por exemplo, no PDA Zaurus da Sharp e no *software* fornecido para a imprensa, candidatos e partidos brasileiros acompanharem as apurações das eleições do mesmo país, em tempo real (Lozano, 2008).

Esta apresenta três modos principais de funcionamento – "*in-memory*", "*standalone*" e "*cliente/server*" – sendo as ligações feitas através de um *driver* JDBC. Caracteriza-se por ser um SGBD pequeno e rápido, que suporta tabelas em disco ou em memória (Azenha, 2006). Relativamente aos modos de funcionamento, têm as seguintes características:

- *In-Memory*

  É possível executar o HSQL num modo em que a base de dados não é persistente e existe unicamente em memória. Como nenhuma informação é escrita para o disco, este modo é recomendado apenas para processamento interno de dados, em "*applets*" ou em aplicações muito específicas (Azenha, 2006).

- *Standalone*

  Este modo lança o motor da BD como parte da aplicação na mesma JVM. Para a maioria das aplicações, este modo poderá ser o mais rápido pois os dados não precisam de ser convertidos e enviados através da rede. A desvantagem consiste

em tornar-se impossível neste modo ligar à BD fora da aplicação que lhe está a aceder (Azenha, 2006).

- *Client/Server*

  O motor encontra-se a correr numa JVM, à espera de ligações de programas na mesma máquina ou de máquinas ligadas na rede. Os clientes ligam-se através de um driver HSQL JDBC. Dependendo do modo de servidor, é possível disponibilizar acessos até 10 bases de dados, sendo estas especificadas na altura do arranque (Azenha, 2006).

Como se pode observar na Figura 29, quer as ferramentas que vêm com o HSQL (interfaces gráfica e ferramentas de gestão/exploração de dados), quer os programas terceiros, acedem ao motor da BD através da interface JDBC. Esta por sua vez permite diferentes tipos de uso: ligações HTTP via Internet usando um servidor *web* ou uma *servlet*; *standalone* em que a aplicação e a BD estão na mesma JVM; ligação TCP/IP numa intranet em que se recorre ao uso de um servidor dedicado (Azenha, 2006).

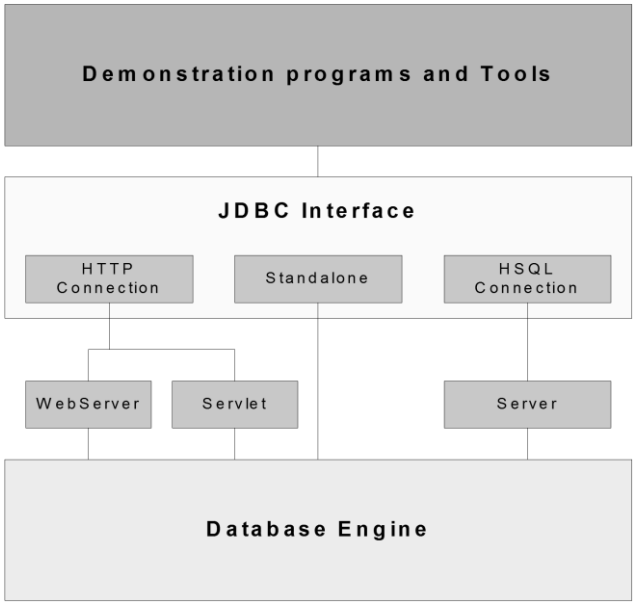


Figura 29 - Arquitetura HSQLDB (Azenha, 2006).

4.1.3.1 Suporte físico

Quando o servidor é lançado ou quando é estabelecida uma ligação uma BD, é sempre criada uma nova BD vazia se ainda não existir nenhuma no caminho especificado. Assim sendo, caso seja cometido um erro a especificar o caminho, é estabelecida à

mesma uma ligação, podendo o utilizador não se aperceber do facto e estranhar por ver uma BD vazia. Para evitar isto, é possível definir a propriedade "ifexists = true" no ficheiro ".properties" para permitir que a ligação seja feita apenas a uma BD que já exista, evitando assim a criação de uma nova BD vazia. Caso esta não exista, é lançada uma exceção (Azenha, 2006).

Nos casos em que o HSQL se executa em modo "*standalone*" ou "*server*", cada base de dados irá consistir num conjunto de 2 a 5 ficheiros todos com o mesmo nome, mas com diferentes extensões, localizados na mesma diretoria da BD. Por exemplo, uma BD com o nome "teste", consistiria nos seguintes ficheiros (Azenha, 2006):

- teste.properties – contém configurações genéricas da BD (versão, compatibilidades, tamanho do ficheiro de log, etc.)
- teste.script – contêm a definição em SQL das tabelas, dos dados e de outros objetos da BD, para as tabelas que não são "*cached*"[2]
- teste.log – contém as alterações recentes à BD
- teste.data – contém os dados para as tabelas "*cached*"
- teste.backup – é uma cópia de segurança comprimida do último estado consistente conhecido para o ficheiro ".data"

Se a BD não tiver tabelas "*cached*", os ficheiros "teste.data" e "teste.backup" não existirão. Adicionalmente, o SGBD pode ligar a outros ficheiros de texto formatados, como por exemplo ficheiros CSV[3], existentes em qualquer diretório do disco (Azenha, 2006).

Enquanto a BD "teste" estiver a ser acedida, o ficheiro "teste.log" é usado para guardar as alterações feitas aos dados. Este ficheiro é removido quando se executa o encerramento da BD através do comando "SHUTDOWN", ou seja, as operações são guardadas no ficheiro ".script" ou ".data" conforme o conteúdo e as propriedades da BD. Caso isso não se verifique (se terminar devido a uma falha), o ficheiro é usado da próxima vez que a BD for iniciada para repor o último estado consistente. Existe ainda um ficheiro teste.lck que é usado para assinalar que a BD está aberta, e que é eliminado após um comando de "SHUTDOW" (Azenha, 2006).

[2] Cached porque quando os dados que se encontram em disco no ficheiro ".data" são necessários, o SGBD carrega-os para memória (Azenha, 2006).
[3] CVS significa "*Comma Separated Values*". É um formato de ficheiro usado para portabilidade de BDs. Cada linha representa um túplo e cada campo encontra-se separado por um ";" (Azenha, 2006).

4.1.3.2 Estrutura e implementação das tabelas

O HSQL define vários tipos de tabelas, as tabelas temporárias (TEMP) e 3 tipos de tabelas persistentes: em memória (MEMORY), em disco (CACHED) e de texto (TEXT).

- Tabelas temporárias

  Duram enquanto a ligação está ativa (Azenha, 2006).

- Tabelas em memória

  São lidos todos os dados existentes no ficheiro ".script" sempre que a BD é iniciada. Quando é executado o comando de SHUTDOW, todos os dados são escritos novamente de memória para o ficheiro ".script". As vantagens deste tipo de tabelas sobre as tabelas em disco são tempos de acesso menores. No entanto, este tipo de tabelas só é adequado para pequenos conjuntos de dados (Azenha, 2006).

- Tabelas em disco

  Os dados só são lidos do ficheiro ".data" quando a tabela é acedida. Assim, só são mantidos em memória parte dos dados e dos índices, donde vem o nome deste tipo de tabelas – CACHED (a quantidade de dados a serem mantidos em cache é configurável). Quando se executa o comando SHUTDOW, os dados em memória que sofreram alterações são escritos para disco e são efetuadas cópias de segurança de todas as tabelas em disco. Este tipo de tabelas são ideais para grandes conjuntos de dados, visto que se acelera o acesso aos mesmos recorrendo à cache, ao mesmo tempo que permite a manipulação de grandes conjuntos de dados que não caberiam todos em memória usando tabelas do tipo MEMORY (Azenha, 2006).

- Tabelas de texto

  Usa ficheiros CSV como fonte de dados. O uso de memória é mais eficiente pois assenta no carregamento para memória da totalidade dos índices e apenas uma pequena parte dos dados. Têm também a vantagem da portabilidade e legibilidade do formato, que praticamente todas as folhas de cálculo, demais

SGBDS e processadores de texto podem ler e modificar. Ficam a perder em termos de desempenho para os outros dois tipos de tabelas persistentes, já que cada acesso a dados implica um potencial acesso a disco, com consequentes *overheads*[4] (Azenha, 2006).

Quanto à estrutura das tabelas, estas são implementadas como objetos Java. Conforme se pode visualizar graficamente na Figura 30, cada tabela contém um nome (que se encontra protegido), um conjunto de colunas (cada objeto coluna contém o nome, tipo dados, etc.), um ou mais índices (os túplos só são acessíveis através de índices pelo que estes são indispensáveis), uma cache para gerir o acesso a dados se a tabela for do tipo MEMORY ou CACHED, e restrições de integridade (se existirem chaves estrangeiras). Se o utilizador não criar uma chave primária, são criados automaticamente uma coluna e índices ocultos a fim de possibilitar os acessos ao túplo (Azenha, 2006).

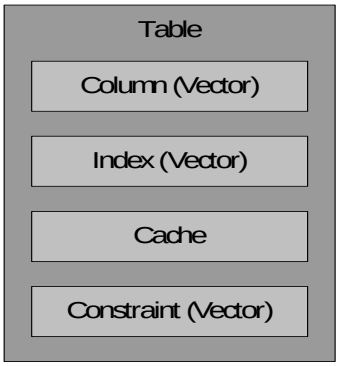


Figura 30 - Estrutura de uma tabela (Azenha, 2006).

4.1.3.3 Armazenamento e acesso a dados

Os dados em HSQL só são acessíveis através de índices (Index). Estes índices são uma implementação de uma árvore binária de procura em que cada nó tem um apontador para o nó pai a fim de permitir navegação bidirecional (raiz para as folhas e vice versa), para além do nó esquerdo e nó direito. No índice existe ainda um nó raiz que se encontra ligado aos restantes nós através de referências a objetos Java ou a apontadores de ficheiros, dependendo da implementação do nó. Cada nó está ligado a um túplo que contém vetores de objetos Java os quais correspondem aos dados do mesmo. O nó raiz

[4] *Overhead* é um custo adicional em processamento ou armazenamento que, como consequência, piora o desempenho de um programa ou de um dispositivo de processamento. São custos adicionais indesejáveis, que deveriam ou poderiam ser evitados (Azenha, 2006).

toma o valor “null” se a tabela estiver vazia. Um objeto “Index” contém ainda informação sobre as colunas da tabela que referencia. (Azenha, 2006). Para melhor perceção, na Figura 31 é apresentada a estrutura de um índice.

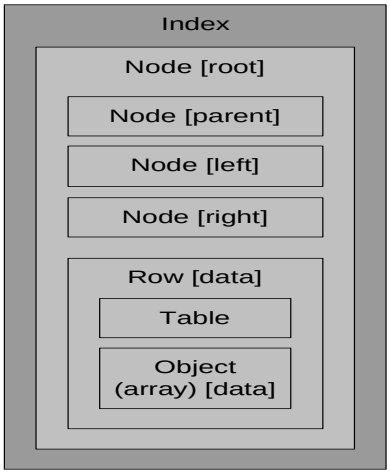


Figura 31 - Estrutura de um índice (Azenha, 2006).

Como se pode verificar na Figura 30, o objeto de uma tabela contém um objeto “Cache” que, recorrendo a mais alguns objetos, gere o acesso aos dados e trata as transferências disco/memória. Esta *cache* funciona como uma “HashTable” em que os túplos se encontram ligados através de listas duplamente ligadas. Quando é necessário espaço para novos dados em memória, os mais antigos são guardados (se necessário) e removidos da *cache* (Azenha, 2006). Com a Figura 32 pode-se ter uma perceção da relação entre os diferentes componentes da *cache*.

Em cada momento existem alguns túplos (Row) na cache, todos eles interligados através da lista duplamente ligada. O *array* “Data” contém em cada *bucket* um apontador para o primeiro túplo (caso exista) mapeado pela função de *hash* para esse *bucket*. Por outro lado, a cache contém também uma ligação para o ficheiro “.data”, o local onde os dados da BD se encontram guardados de forma persistente.

Com esta estrutura torna-se fácil aceder de forma aleatória aos túplos graças à “HashTable” e ainda permite que as inserções e remoções de túplos da *cache* seja relativamente simples (Azenha, 2006).

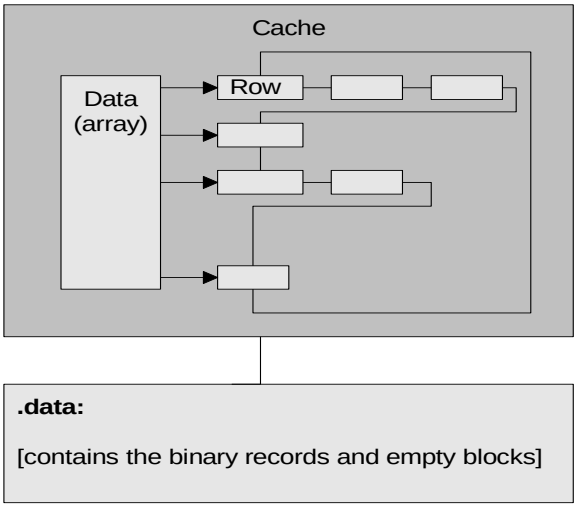


Figura 32 - Arquitetura da cache (Azenha, 2006).

Relativamente aos tipos de dados suportados pelo HSQL, para além dos tipos tradicionais, este apresenta ainda suporte para tipos de dados especiais, os tipos OTHER e OBJECT. Isto torna possível, com auxílio da ferramenta "SQL Tool" do HSQL e seu *buffer*, o armazenamento de dados binários nas colunas, tais como fotografias, ficheiros áudio ou objetos Java serializados. Quando os dados estão no *buffer*, recorrendo ao comando "\bl", podem então ser escritos para uma coluna da BD, recorrendo ao comando "\bp" e ao uso de "prepared statments", como por exemplo (Azenha, 2006):

```
$ \bl /tmp/music.mp3
$ \bp
$ INSERT INTO tMusic (id, stream) VALUES (123, ?);
```

Desta forma, é inserido o ficheiro de áudio MP3, denominado por "music", na tabela "tMusic". Por outro lado, para obter os mesmos dados, recorrendo ao uso de "prepared statments" e do comando "\bd", seria a seguinte forma (Azenha, 2006):

```
$ SELECT stream FROM tMusic WHERE id = 123;
$ \bd /tm/music.mp3
```

4.2 Aplicação do modelo conceptual em HSQLDB

Começou-se por compilar o código fonte do motor de base de dados do HSQL no IDE Eclipse. Após ter o projeto compilável e executável através do respetivo IDE foram feitos alguns testes de CRUD e *debug* dos mesmos de forma a perceber as

potencialidades e comportamento em *background* do respetivo motor. Abaixo seguem todos os passos realizados até à obtenção da solução idealizada.

4.2.1 Criar nova palavra SQL não reservada

À semelhança de outras palavras SQL já existentes (por exemplo, “LTRIM”, “RTRIM” “SYSDATE”, “DATE_ADD”, “TO_CHAR”, entre outras) foi criada uma nova palavra SQL não reservada denominada por “BUNDLE”. A criação desta nova palavra teve como base a criação de um novo *commandSet* na classe “Tokens.java”, da seguinte forma:

```
static final String T_BUNDLE = "BUNDLE";
(…)
static final int BUNDLE = 798;
(…)
commandSet.put(T_BUNDLE, BUNDLE);
```

Figura 33 - Definição de Token e CommandSet "BUNDLE".

Na Figura 34 encontra-se a definição do mapeamento entre o *Token* definido anteriormente e uma nova função.

```
protected static final int FUNC_BUNDLE_STRING = 64;

(…)

customRegularFuncMap.put(Tokens.BUNDLE, FUNC_BUNDLE_STRING);
```

Figura 34 - Mapeamento entre Token e nova função.

Assim, esta palavra não reservada está agora associada a uma função como sendo nativa da própria base de dados. Nesta função pode ser passada a seguinte informação de entrada:

1. Chave, que corresponde ao identificador da legenda que se pretende obter. Este parâmetro é obrigatório.
2. Conjunto de valores a substituir na legenda correspondente à chave indicada anteriormente. Este é opcional e pode conter uma quantidade ilimitada de valores, sendo os mesmos colocados na mensagem sempre por ordem sequencial.
3. *Locale* específico do qual se pretende obter a legenda.

Na seguinte instrução, contida nas classes “FunctionCustom.java” e “FunctionSQL.java”, a função criada é definida com três parâmetros de entrada, sendo que os segundo e terceiro parâmetros são opcionais:

```
case FUNC_BUNDLE_STRING :
        name      = Tokens.T_BUNDLE;
        parseList = optionalTripleParamList;
        break;
```

Figura 35 - Definição da tipologia de parâmetros de entrada da nova função "BUNDLE".

Para este caso, teve que ser declarado um novo tipo de lista de parâmetros, na classe "FunctionSQL.java", o qual permite a inserção de até três parâmetros de entrada, conforme apresentado na Figura 36.

```
static final short[] optionalTripleParamList = new short[] {
      Tokens.OPENBRACKET, Tokens.QUESTION, Tokens.X_OPTION, 2, Tokens.COMMA,
      Tokens.QUESTION, Tokens.X_OPTION, 2, Tokens.COMMA, Tokens.QUESTION,
      Tokens.CLOSEBRACKET
   };
```

Figura 36 - Criação da nova tipologia de parâmetros de entrada.

O segundo parâmetro, para além de ser opcional, é do tipo *Varchar* e pode conter uma quantidade ilimitada de valores separados por vírgulas, sendo os mesmos colocados na mensagem (substituindo o conjunto de caracteres "$$") sempre pela ordem que o programador indicar. Serão substituídos tantos valores quantos "$$" existirem na legenda. No ponto 4.2.1.1 serão apresentados vários exemplos de uso desta função.

O terceiro parâmetro é, também, opcional e do tipo *Varchar* o qual receberá o código do idioma do qual se pretende obter a mensagem (exemplo: "fr_FR" – francês de França). No ponto 4.2.2.1 serão apresentados mais em detalhe exemplos do uso deste parâmetro.

Numa primeira fase desta implementação colocou-se esta função a recorrer aos pacotes de recursos do Java, criando assim dois novos ficheiros de propriedades para teste, contendo cada um deles a seguinte informação:

Tabela 6 - Conteúdo dos Resource Bundles em inglês (usado por defeito) e em português

custom-messages.properties	**custom-messages_pt_PT.properties**

#
#
# @author CatarinaSantos 	
# @version 0.0.1 	
#
# message parts 	
0001=Welcome!
0002=Receipt $$ successfully created.
0003=The receipt $$ has an invalid amount.
0004=The receipt $$ has $$ invalid items.
	#
	#
	# @author CatarinaSantos
	# @version 0.0.1
	#
	# message parts
	0001=Bem-vindo!
	0002=O recibo $$ foi criado com sucesso.
	0003=O recibo $$ tem um montante inválido.
	0004=O recibo $$ tem $$ itens inválidos.

Por outro lado, foi criada no projeto uma área específica para o tratamento da internacionalização, conforme a Figura 37.

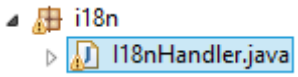


Figura 37 - Novas classes para tratamento da internacionalização

A classe "I18nHandler.java" é a responsável por manipular informação relativa à internacionalização desta base de dados. Nesta primeira fase, esta tem como principal tarefa a consulta, aos pacotes de recursos do Java, das chaves que recebe como entrada e substituir eventuais valores que venham, também, como parâmetro de entrada.

Serviram de inspiração algumas das funcionalidades das tecnologias Java e ZF para desenvolver esta fase, nomeadamente:

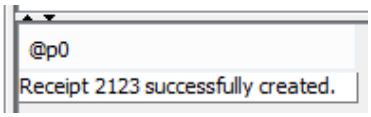
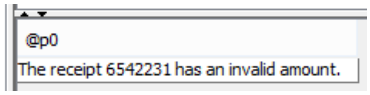
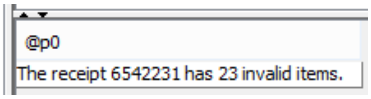
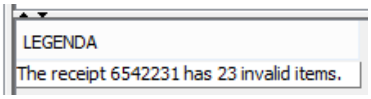
- Existência de uma função para obter o conteúdo localizável, neste caso mensagens de texto. (Java e ZF)
- Mensagens de texto dinâmico com possibilidades de substituição de valores no momento da chamada. (ZF)
- Determinação do *locale* especifico a ser chamado, não usando assim o padrão. (Java e ZF)

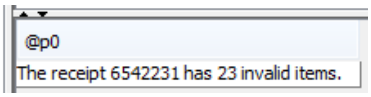
4.2.1.1 Resultados

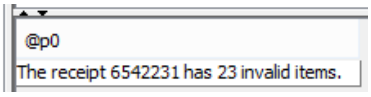
Com estes primeiros desenvolvimentos já se concretizou uma parte do modelo idealizado, que consiste na existência de uma nova palavra SQL não reservada como nativa da base de dados. Na Tabela 7 são listados os resultados dos primeiros testes realizados.

Tabela 7 - Atividades experimentais (Locale = en_GB)

Comando	**Resultado**	**Observação**
`CALL BUNDLE('0001');`		Chamada da nova função para devolver a mensagem 0001.

Comando	Resultado	Observação
`CALL BUNDLE('0002', '2123');`		Chamada da nova função para devolver a mensagem 0002, substituindo o valor do segundo parâmetro pelo conjunto de caracteres "$$" contido na legenda.
`CALL BUNDLE('0003', '6542231, 23');`		A mensagem 0003 contem apenas um conjunto de caracteres "$$". Desta forma, mesmo que se coloque vários valores no segundo parâmetro de entrada ele colocará apenas o primeiro valor na legenda.
`CALL BUNDLE('0004', '6542231, 23');`		Usando os mesmos valores para a mensagem 0004 (contem dois conjuntos "$$"), ambos já serão colocados sequencialmente nos respetivos lugares.
`SELECT TOP 1 BUNDLE('0004', '6542231, 23') as LEGENDA FROM customer;`		Sendo uma função, é possível usá-la também em *queries* SQL.

`DECLARE variavel_teste VARCHAR(255);`
 	
`SET variavel_teste = BUNDLE('0004', '6542231, 23');`
 	
`CALL variavel_teste;`
		No seguimento do ponto anterior, também é possível guardar o seu valor numa variável do tipo *Varchar*.

`CREATE PROCEDURE procedimento_teste(OUT legenda varchar(100))`
`MODIFIES SQL DATA`
`BEGIN ATOMIC`
`SET legenda = BUNDLE('0004', '6542231, 23');`
`END;`
`----------`
`DECLARE param VARCHAR(100);`
`CALL`
		No seguimento do ponto anterior, também é possível invocar esta função dentro de procedimentos.
		Neste caso o procedimento "procedimento_teste" tem um parâmetro de *output* no qual é guardado a legenda 0004.

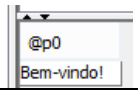
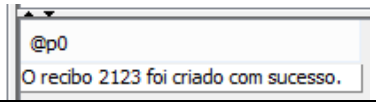
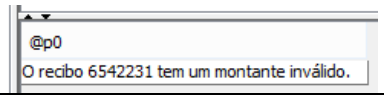
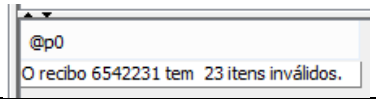
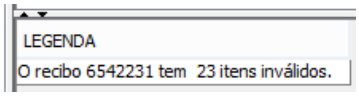
Comando	Resultado	Observação

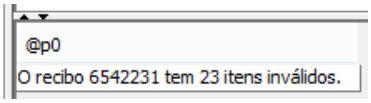
`procedimento_teste(param);`
 	
`CALL param;`		

Nestes primeiros testes o idioma de sistema usado foi o inglês da Grã-Bretanha (en_GB), assim sendo, como não existe nenhum ficheiro de propriedades "custom-messages_en_GB" as legendas usadas foram as por defeito "custom-messages" que, por sinal, têm as legendas em inglês.

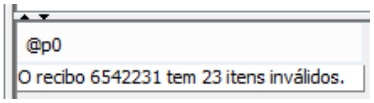
De seguida, alterou-se o idioma para português de Portugal (pt_PT) e efetuou-se os mesmos testes obtendo, assim, os resultados da Tabela 8.

Tabela 8 - Atividades experimentais (Locale = pt_PT)

Comando	Resultado	Observação
`CALL BUNDLE('0001');`		
`CALL BUNDLE('0002', '2123');`		
`CALL BUNDLE('0003', '6542231, 23');`		
`CALL BUNDLE('0004', '6542231, 23');`		
`SELECT TOP 1 BUNDLE('0004', '6542231, 23') as LEGENDA FROM customer;`		

`DECLARE variavel_teste VARCHAR(255);`
 	
`SET variavel_teste = bundle('0004', '6542231, 23');`
 	
`CALL variavel_teste;`
		

Comando	Resultado	Observação

`DECLARE param VARCHAR(100);`
`CALL procedimento_teste(param);`
`CALL param;`
		Neste teste já não foi necessário criar novamente o procedimento "procedimento_teste", pois o mesmo já foi criado nos testes anteriores.

Verificou-se que todas as mensagens foram retornadas em português, uma vez que existe um ficheiro de propriedades "custom-messages_pt_PT" com as respetivas mensagens em português.

Na figura abaixo encontra-se um esboço da arquitetura atual desta solução.

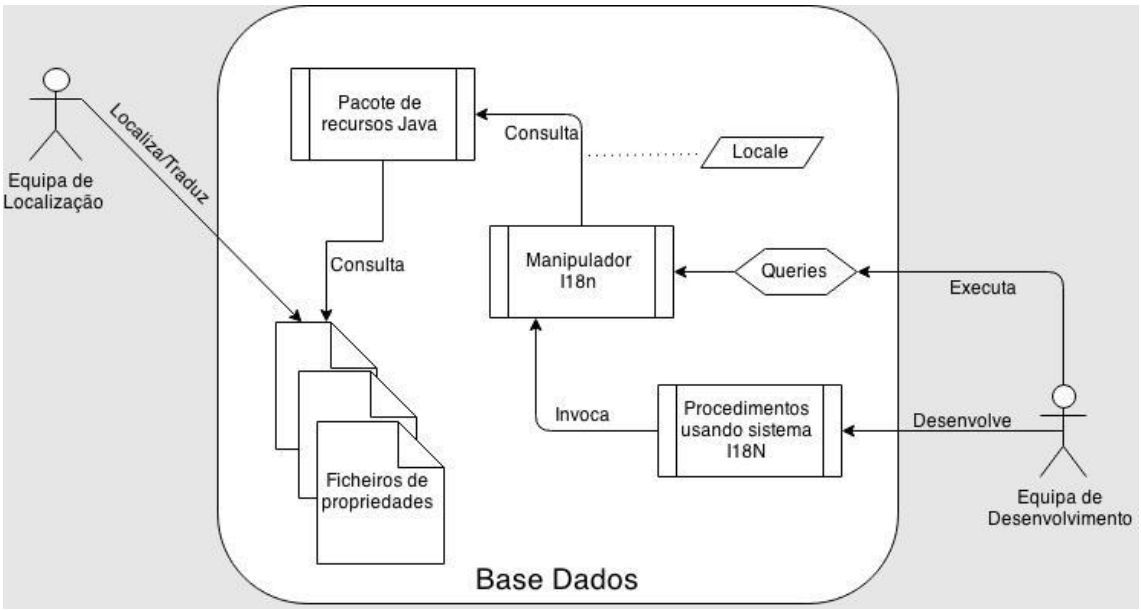


Figura 38 - Estado atual do modelo

Com os desenvolvimentos atuais já se obteve uma parte do modelo definido no ponto 3 desta dissertação, conforme se pode verificar a verde na Figura 39.

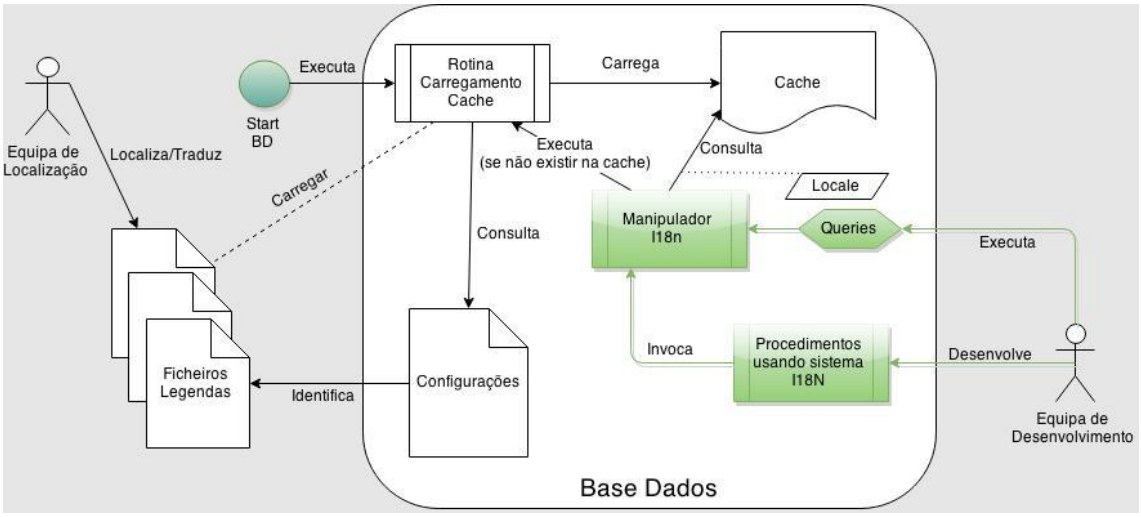


Figura 39 - Primeiro passo no modelo I18n em base de dados.

4.2.2 Consultar ficheiros externos

No ponto 4.2.1 utilizou-se os pacotes de recursos do Java para armazenar e aceder às legendas, mas desta forma sempre que há uma alteração dos ficheiros de propriedades estas só são assumidas após a aplicação em questão ser reiniciada. No caso de uma base de dados é importante que a mesma esteja indisponível o mínimo de tempo possível, logo, o objetivo é poder efetuar alterações sobre os ficheiros, como por exemplo adicionar novas legendas, e estas estarem imediatamente acessíveis pela base de dados sem ser necessário reiniciar a mesma.

Neste sentido, substituíram-se os pacotes de recursos do Java pela consulta direta a ficheiros externos, pesquisando a chave solicitada em cada uma das linhas dum ficheiro. É de notar que, na passagem dos pacotes de recursos Java para os ficheiros externos, a lógica e estrutura tanto do nome do ficheiro como do seu conteúdo foram mantidas (à exceção da extensão do ficheiro). A seleção do ficheiro a consultar depende igualmente do *locale*. O método criado para este efeito foi o `getMessageInFile` na classe `I18nHandler.java`.

Logo à partida os ficheiros externos e o seu conteúdo têm que obedecer a três regras, que são as seguintes:

1. Os ficheiros devem ter extensão “.txt”.

2. O nome de cada um dos ficheiros deve obedecer à seguinte expressão “custom-messages[_LOCALE_CODE]”, em que o *locale code* é opcional. Quer isto dizer que “custom-massages.txt” é o nome do ficheiro padrão de mensagens.
3. Relativamente às mensagens contidas dentro do mesmo, caso estes tenham que suportar valores variáveis, o sítio onde os mesmos serão substituídos tem que ser identificado com a sequência de símbolos “$$”. Cada sequência destas corresponde a um valor diferente que será substituído.

Para além destas características, há mais duas propriedades relativas aos ficheiros que, estas sim, podem ser alteradas pelo programador em tempo de execução da base de dados. São estas propriedades a localização (pasta no sistema de ficheiros local) dos ficheiros e o *locale* a que corresponde o ficheiro de mensagens padrão.

A localização no computador, por defeito, dos ficheiros *Bundle* é a mesma pasta onde se encontra instalado o respetivo motor de base de dados, sendo esta também parametrizável através da seguinte instrução SQL:

```
SET BUNDLE DIRECTORY [directory_path];

-- Exemplo:

SET BUNDLE DIRECTORY '.\I18n\';
```

Figura 40 – Alterar a diretoria onde são consultados os ficheiros externos.

O caminho definido tanto pode ser relativo como absoluto. É de notar que o caminho relativo é sempre a partir da pasta onde se encontra instalado o motor de base de dados.

Por outro lado, é possível parametrizar, de igual forma, a que idioma corresponde o ficheiro *bundle* padrão (custom-bundle.txt), através da seguinte instrução SQL:

```
SET BUNDLE FILE_DEFAULT_LOCALE [locale_code];

-- Exemplo:

SET BUNDLE FILE_DEFAULT_LOCALE 'pt_PT';
```

Figura 41 - Alterar o idioma relativo ao ficheiro de mensagens padrão (custom_messages.txt).

Ambas as alterações não implicam reiniciar a base de dados, antes pelo contrário, pois após se reiniciar a mesma as propriedades em questão voltam a ter o valor por defeito (“.\” e “en_GB”, respetivamente).

Para este efeito foram criados dois novos *tokens*, na classe “Tokens.java”:

```
static final String T_DIRECTORY           = "DIRECTORY";
static final String T_FILE_DEFAULT_LOCALE  = "FILE_DEFAULT_LOCALE";
```

Figura 42 - Definição dos tokens "DIRECTORY" e "FILE_DEFAULT_LOCALE".

E dois novos tipos de instrução, nas classe “StatementType.java”, que são posteriormente definidas no grupo de propriedades da base de dados em si:

```
int SET_DATABASE_BUNDLE_DIRECTORY          = 1057;
int SET_DATABASE_BUNDLE_FILE_DEFAULT_LOCALE = 1058;
```

Figura 43 - Definição das variantes do comando "SET BUNDLE".

Quando é executado algum destes dois comandos (começado por “SET BUNDLE”) há um ponto de decisão em que se verifica qual o *token* seguinte, conforme exemplificado abaixo.

```
StatementCommand compileSetBundleProperty() {

switch (token.tokenType) {

case Tokens.DIRECTORY : {
      type  = StatementTypes.SET_DATABASE_BUNDLE_DIRECTORY;
      break;
}
case Tokens.FILE_DEFAULT_LOCALE : {
      type  = StatementTypes.SET_DATABASE_BUNDLE_FILE_DEFAULT_LOCALE;
      break;
}
(…)
```

Figura 44 - Definição do tipo de comando consoante o token usado.

Dependendo do *token* seguinte, é atribuído um tipo de instrução diferente que, por sua vez, será posteriormente verificada no método “getResult()” da classe “StatementCommand.java” e executado o respetivo método, conforme se pode verificar abaixo.

```
case StatementTypes.SET_DATABASE_BUNDLE_DIRECTORY : {
        String dir = parameters[0].toString();
        I18nHandler.setMsgPropsDir(dir);
        return Result.updateZeroResult;
}
case StatementTypes.SET_DATABASE_BUNDLE_FILE_DEFAULT_LOCALE : {
        String locale = parameters[0].toString();
        I18nHandler.setLocaleDefault(locale);
        return Result.updateZeroResult;

}
```

Figura 45 - Execução do respetivo método consoante o tipo de comando, devolvendo o resultado.

É de notar que a alteração destas duas propriedades é mantida apenas em memória, por isso, assim que a base de dados é reiniciada os valores das mesmas voltam aos valores padrão.

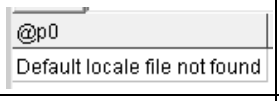
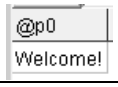
Nesta fase a tecnologia que serviu de inspiração para definir a estrutura do nome dos ficheiros externos e do seu conteúdo foi a tecnologia Java.

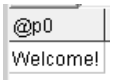
4.2.2.1 Resultados

Com os desenvolvimentos desta etapa já é possível obter uma mensagem diretamente do ficheiro externo, o que permite poder alterar a mensagem no ficheiro e esta encontrar-se, de imediato, disponível para consulta após as alterações sem que, para isso, seja necessário reiniciar a base de dados. Da mesma forma, também, é possível alterar a localização dos ficheiros externos e/ou o idioma do ficheiro padrão.

Na Tabela 9 é apresentada a concretização de alguns testes de validação desta etapa.

Tabela 9 - Atividades experimentais (ficheiros externos)

Comando	**Resultado**	**Observação**
`CALL BUNDLE('0001');`		Iniciou-se com os ficheiros na diretoria ".\".
`CALL BUNDLE('0001');`		Moveram-se os ficheiros para a pasta ".\I18n\" antes de executar novamente o comando.
`SET BUNDLE DIRECTORY '.\I18n\';`		Executou-se a instrução SQL para definir a diretoria dos ficheiros *bundle* externos.
`CALL BUNDLE('0001');`		Voltou a encontrar o ficheiro e a devolver a mensagem corretamente.
`SET BUNDLE FILE_DEFAULT_LOCALE`		Executou-se a instrução SQL para

Comando	Resultado	Observação
`'pt_PT';`		definir qual o idioma do ficheiro *bundle* padrão como português de Portugal (pt_PT). É de notar que o ficheiros padrão (custom-messages.txt) contem as mensagem em inglês.
`CALL BUNDLE('0001', '','pt_PT');`		Como o idioma do ficheiros padrão está definido como "pt_PT" e o ficheiros padrão contém as mensagens em inglês pode constatar-se que, forçando a instrução SQL a ir buscar a mensagem em "pt_PT", a que é devolvida é em inglês, mesmo existindo um ficheiro "custom-messages_pt_PT"

Na Figura 46 é apresentado o estado atual do modelo de internacionalização em bases de dados *open source*, estando neste momento duas etapas concluídas, nomeadamente, a criação de novas instruções SQL e respetivo manipulador e a consulta de ficheiros externos.

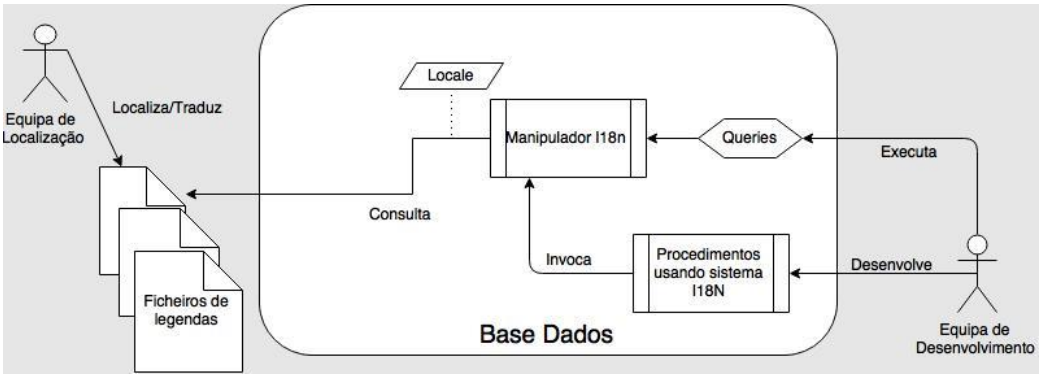


Figura 46 - Estado atual do modelo.

Do modelo conceptual definido, já se encontram concretizados os elementos indicados a verde na Figura 47.

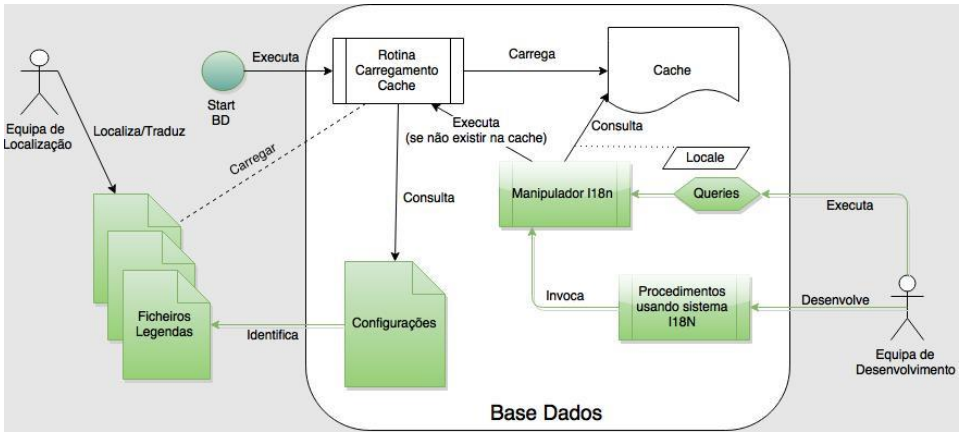


Figura 47 - Segundo passo no modelo I18n em base de dados.

4.2.3 Rotina para carregamento da *cache*

Uma vez que a consulta constante a ficheiros externos requer algum esforço de processamento adicional, houve a necessidade de colocar as mensagens num local mais acessível, onde o esforço de consulta seja menor. Desta forma, à semelhança do ZF, optou-se por carregar em memória todas as mensagens, caso existam, assim que a base de dados é iniciada. Para este efeito, foi criada a classe "I18nCache.java" a qual representa a *cache* de internacionalização da base de dados.

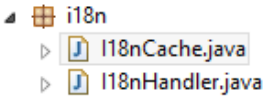


Figura 48 - Nova classe correspondente à cache.

Esta *cache* consiste, basicamente, num *HashMap* sincronizado constituído por pares <*key*,*value*> (neste caso, `<String, Object>`), onde o primeiro elemento é a chave da mensagem e o segundo elemento é o valor da mensagem.

Na classe "`I18nHandler`" existe o método "`initCache`" que é chamado pelo método "`reopen`" da classe "`Database`", o qual é usado no arranque da base de dados.

Ao serem carregadas, as mensagens, são guardadas no *HashMap* de acordo com o apresentado na Tabela 10.

Tabela 10 - Mensagens guardadas na I18nCache.

Chave	**Valor**
[LOCALE][CODIGO_MENSAGEM]	[MENSAGEM]

Chave	Valor
Exemplo	
en_GB0001	Welcome!
en_GB0002	Receipt $$ successfully created.
pt_PT0001	Bem-vindo!
pt_PT0002	O recibo $$ foi criado com sucesso.
…	

De acordo com a Figura 49, para os ficheiros identificados com o *locale* (ex: custom-messages_pt_PT.txt) este é o inserido como prefixo da chave. No caso do ficheiro padrão de mensagens (custom-messages.txt) o *locale* usado como prefixo é o da propriedade *"localeDefault"*, a qual identifica o *locale* do ficheiro padrão.

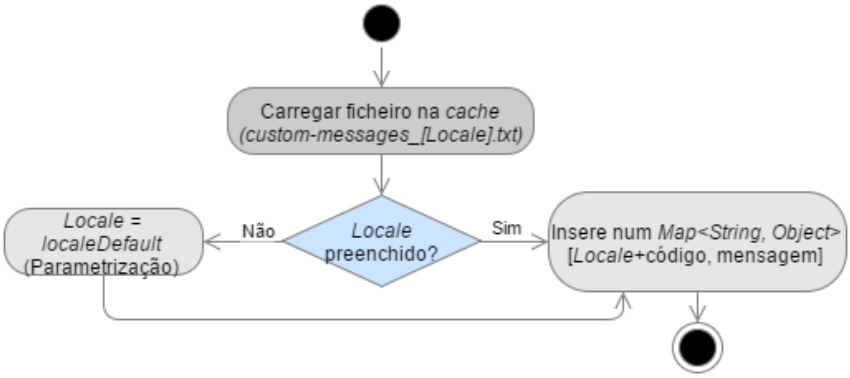


Figura 49 - Lógica de carregamento das mensagens (chave-valor) na cache.

Uma vez feito o carregamento inicial da *cache* é possível fazer, posteriormente, *reset* à mesma caso seja necessário. Para o efeito, à semelhança da palavra "BUNDLE", foi criada uma nova palavra denominada por "RESETBUNDLE", conforme a Figura 50.

```
static final String T_RESET_BUNDLE = "RESETBUNDLE";
(…)
static final int RESET_BUNDLE = 799;
(…)
commandSet.put(T_RESET_BUNDLE, RESET_BUNDLE);
```

Figura 50 - Definição de Token e CommandSet "RESETBUNDLE".

A esta nova palavra foi associada uma nova função, de acordo com a figura abaixo.

```
protected static final int FUNC_BUNDLE_RESET = 65;

(…)

customRegularFuncMap.put(Tokens.RESET_BUNDLE, FUNC_BUNDLE_RESET);
```

Figura 51 - Mapeamento entre o Token e nova função.

Foi definido para a respetiva função o tipo de lista de parâmetros de entrada já existente, o qual permite à função receber apenas um parâmetro opcional, tal como é apresentado na Figura 52.

```
case FUNC_BUNDLE_RESET :
        name      = Tokens.T_RESET_BUNDLE;
        parseList = optionalSingleParamList;
        break;
```

Figura 52 - Definição da tipologia de parâmetros de entrada da nova função "RESETBUNDLE".

Esta funcionalidade é útil no caso de alguma das mensagens, já existente na cache, seja modificada, isto é, mantendo a mesma chave e modificando apenas o seu valor. Nestes casos, pode-se fazer *reset* apenas da mensagem em questão, conforme apresentado na Figura 53.

```
CALL RESETBUNDLE('0001');
```

Figura 53 - Remover a mensagem "0001" da cache.

Por outro lado, pode-se fazer *reset* da cache por inteiro não especificando nenhum código de mensagem, tal como na figura abaixo.

```
CALL RESETBUNDLE();
```

Figura 54 - Remover todas as mensagens da cache.

Na verdade o que acontece com a chamada desta função, no caso da Figura 53, é a eliminação da mensagem de código "0001" do *HashMap* em memória, para todas as linguagens carregadas. No caso da Figura 54, são eliminadas todas as mensagens do *HashMap*, sem exceção.

A inspiração para a criação desta *cache* i18n nesta fase foi a tecnologia ZF, a qual contém o componente Zend_Cache que mantem carregadas as fontes de tradução para consulta. Desta forma, as consultas tornam-se mais rápidas.

4.2.3.1 Resultados

Foram feitos alguns testes para verificar que a consulta das mensagens na cache, geralmente, é mais rapidamente processada que a consulta direta ao ficheiro externo. Para fazer esta verificação colocou-se mensagens de output mostrando o tempo em nano segundos antes e depois das invocações dos métodos de pesquisa na cache e nos ficheiros externos, tal como é mostrado na Figura 55. As regras aplicadas à lógica desta consulta serão abordadas em detalhe no ponto 4.2.4.

```
System.out.println("Get Cache init:" + System.nanoTime());
result = (String)I18nCache.getInstance().get(locale + key);
System.out.println("Get Cache end:" + System.nanoTime());
   if (result == null){
       System.out.println("Get file init:" + System.nanoTime());
       result = getMessageInFile(key, locale);
       System.out.println("Get file end:" + System.nanoTime());
       (…)
```

Figura 55 - Log técnico para estudo de performance.

Na Tabela 11 estão registados os resultados dos tempos obtidos após os testes de performance realizados.

Tabela 11 - Testes de performance das consultas à cache e ficheiros.

Output	**Diferença de tempo (nano segundos)**	**Diferença de tempo (%)** (Tempo da cache em relação aos ficheiros)
1ª Execução		
Get Cache init: 29851293457147	63721	20,08
Get Cache end: 29851293520868		
Get file init: 29851293567054	317320	
Get file end: 29851293884374		
2ª Execução		
Get Cache init: 30417357027512	80399	21,61
Get Cache end: 30417357107911		
Get file init: 30417357132287	372060	
Get file end: 30417357504347		

Output	**Diferença de tempo (nano segundos)**	**Diferença de tempo (%)** **(Tempo da cache em relação aos ficheiros)**
3ª Execução		
Get Cache init: 30497047266904	94940	29,96
Get Cache end: 30497047361844		
Get file init: 30497047399477	316892	
Get file end: 30497047716369		

Após três execuções aleatórias para obtenção de uma mensagem em especifico (*call bundle('0001');*) verificou-se que o tempo de acesso à *cache* é entre 20 a 30 por cento do tempo de acesso aos ficheiros externos. Tratando-se de tempo em nano segundos, a diferença é impercetível a "olho nu", mas em processamento de grandes quantidades de informação, por parte da base de dados, esta diferença acumulada pode causar algum impacto na sua performance.

Uma vez que as mensagens são carregadas na *cache*, assim que a base de dados inicia, para uma consulta mais direta, foi necessário criar a possibilidade de limpar a mesma (RESETBUNDLE) caso haja alterações às mensagens existentes, sem que seja necessário reiniciar a base de dados para as mensagens serem carregadas já com as alterações. Na Tabela 12 são apresentados os resultados dos testes efetuados a esta funcionalidade de limpeza da *cache*. Com as mesmas mensagens de output dos tempos consegue-se perceber quando é que a consulta é feita aos ficheiros ou não, pois a consulta dos ficheiros só acontece caso a mensagem pesquisada não exista na *cache*.

Tabela 12 - Testes de limpeza da cache de mensagens

Comando	**Log do servidor**	**Observação**
call bundle('0001');	Get Cache init:1939719670941 49 Get Cache end:193971967434990	Foi impresso na consola apenas as mensagens de *output* referente à pesquisa na *cache*, uma vez que a mensagem "0001" já lá existia.
call resetbundle('0001');		Foi limpa a mensagem "0001" da *cache*.

Comando	Log do servidor	Observação
call bundle('0001');	Get Cache init:193971967094149 Get Cache end:193971967434990 Get Cache init:194008046680591 Get Cache end:194008046818723 Get file init:194008046964125 Get file end:194008047315657	Como a mensagem "0001" não foi encontrada na *cache*, foi feita a consulta nos ficheiros externos e, ao ser encontrada, a mesma foi reinserida imediatamente na *cache*.
call bundle('0001');	Get Cache init:193971967094149 Get Cache end:193971967434990 Get Cache init:194008046680591 Get Cache end:194008046818723 Get file init:194008046964125 Get Cache init:194404815054538 Get Cache end:194404815220468	Uma vez já recarregada na execução anterior, a mensagem "0001" é encontrada e imediatamente retornada na consulta da *cache*, dispensando assim a consulta aos ficheiros externos.

Para alem destes testes, executando o comando "call resetbundle();" (sem parâmetros de entrada) a cache é limpa por completo e, a partir daí, as mensagens são recarregadas uma a uma à medida que vão sendo solicitadas. A menos que a base de dados seja reiniciada e sejam todas as mensagens carregadas novamente.

Com estes últimos testes pôde-se validar, assim, a funcionalidade de limpeza de cache. Na Figura 56 é apresentado o estado atual do modelo, segundo o que já foi desenvolvido, testado e documentado.

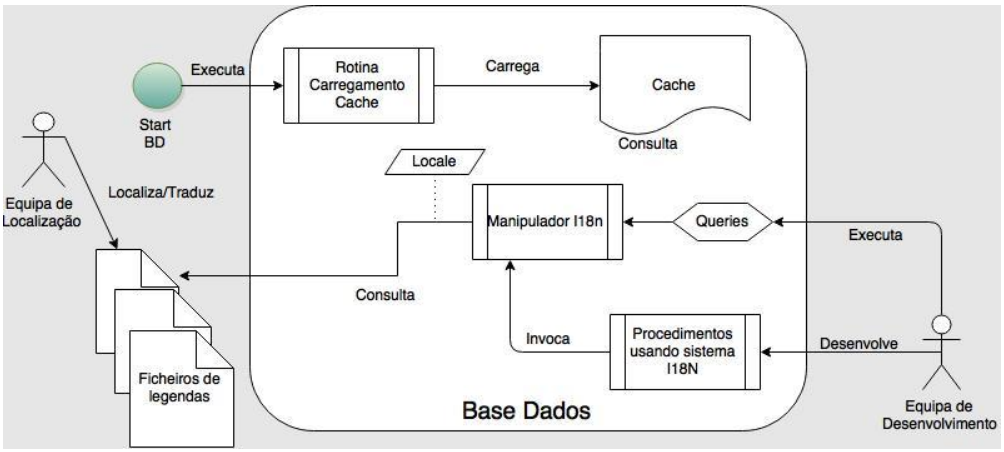


Figura 56 - Estado atual do modelo.

Atualizando novamente o modelo definido no ponto 3 desta dissertação, encontra-se assinalado a verde, na figura abaixo, os elementos que já estão concluídos, no contexto do mesmo.

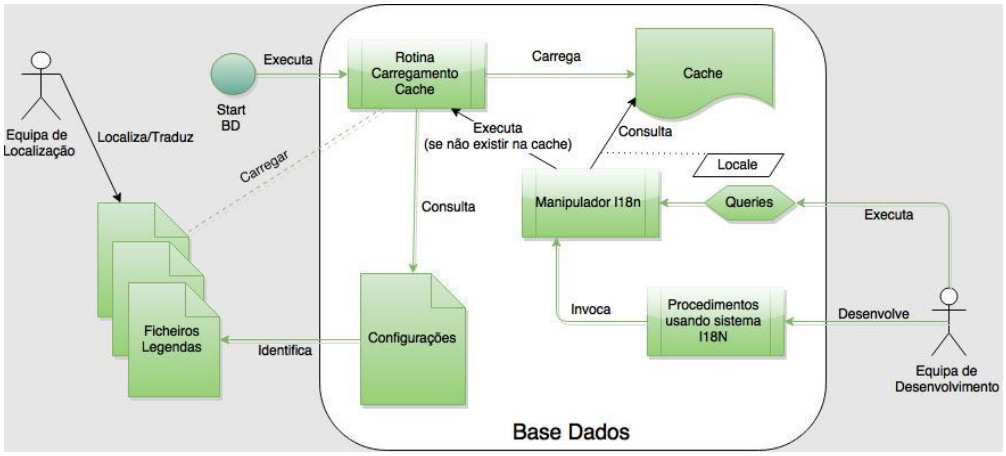


Figura 57 - Terceiro passo no modelo I18n em base de dados.

4.2.4 Consulta da *cache* pelo manipulador I18n

Tal como referido anteriormente, o módulo desenvolvido é constituído por duas classes, nomeadamente, "I18nHandler.java" e "I18nCache.java", em que a primeira corresponde ao manipulador I18n e a segunda representa a *cache* onde são carregadas todas as mensagens em memória durante toda a execução da base de dados. Neste caso, a lógica de consulta das mensagens encontra-se no manipulador. Lógica esta que é ilustrada de seguida, através de um diagrama de atividade e de sequência (Figura 58 e Figura 59), no sentido de se perceber mais facilmente através dos respetivos elementos gráficos.

O ponto de partida do diagrama de atividade que se segue é um pedido de uma mensagem, ou seja, uma execução da instrução "BUNDLE" para uma mensagem específica. Esta execução é independentemente da sua origem, isto é, programador (através de *queries*), procedimento ou outra.

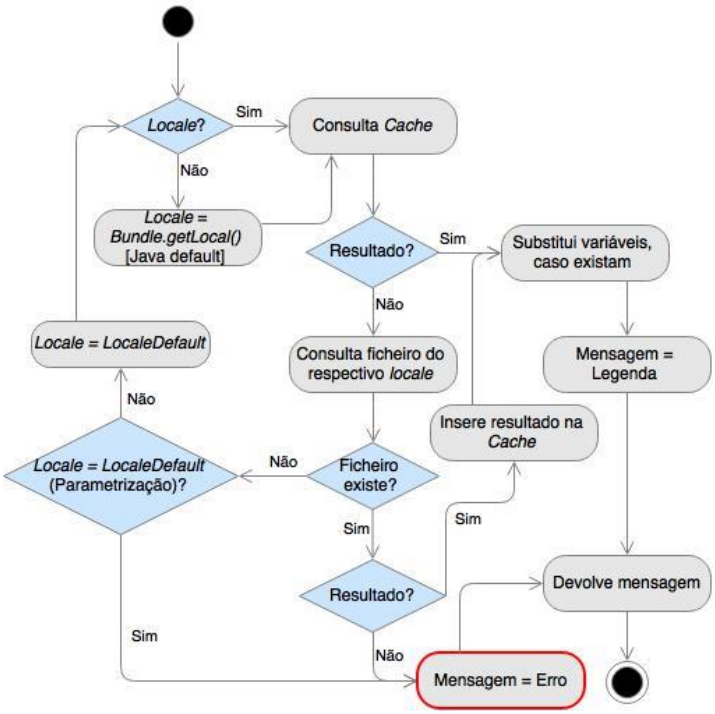


Figura 58 - Consulta de mensagens pelo manipulador I18n – Diagrama de atividade.

É sempre validado à cabeça se o *locale* é fornecido como um dos parâmetros de entrada. Caso não o seja, é usado o *locale* padrão da base de dados em uso. Daqui para a frente, este *locale* é sempre usado tanto para pesquisar a mensagem na *cache* (prefixo da chave), como para pesquisar a mensagem no ficheiro do respetivo idioma (sufixo do nome do ficheiro).

A mensagem é sempre pesquisada em primeiro lugar na *cache* e só se não existir é que a mesma vai ser pesquisada nos ficheiros externos. Caso esta exista nos ficheiros, a mesma é inserida imediatamente na *cache* antes de ser retornada, para numa próxima consulta esta já lá existir. Se o *locale* a ser pesquisado não corresponder ao padrão (propriedade FILE_DEFAULT_LOCALE) e não existir ficheiro para o respetivo idioma, é feita novamente a pesquisa de início para o idioma padrão (*localeDefault*) seguindo, assim, os mesmos passos. Desta forma, numa pesquisa é devolvido erro em apenas dois casos, sendo estes os seguintes:

- Existe o ficheiro externo para o idioma pesquisado, mas o mesmo não contem a mensagem solicitada. Erro: *"02000 no data"*.
- Não existe ficheiro de mensagens padrão (custom-messages.txt) caso a pesquisa necessite de recorrer a este ficheiro. Erro: *"Default locale file not found"*.

Por último, uma vez obtida a mensagem, quer pela *cache* ou por ficheiro, antes da mesma ser retornada são substituídas na mensagem todas as variáveis possíveis, caso

existam, isto é, caso seja enviada alguma lista de valores como segundo parâmetro de entrada da instrução "BUNDLE" e existam *tags* variáveis ($$) incluídas na mensagem a ser retornada.

Na figura seguinte, é apresentada exatamente a mesma lógica do diagrama de atividade, mas em formado de diagrama de sequência, o qual pretende ser mais detalhado relativamente à interação entra as classes constituintes do módulo I18n e ao algoritmo em si.

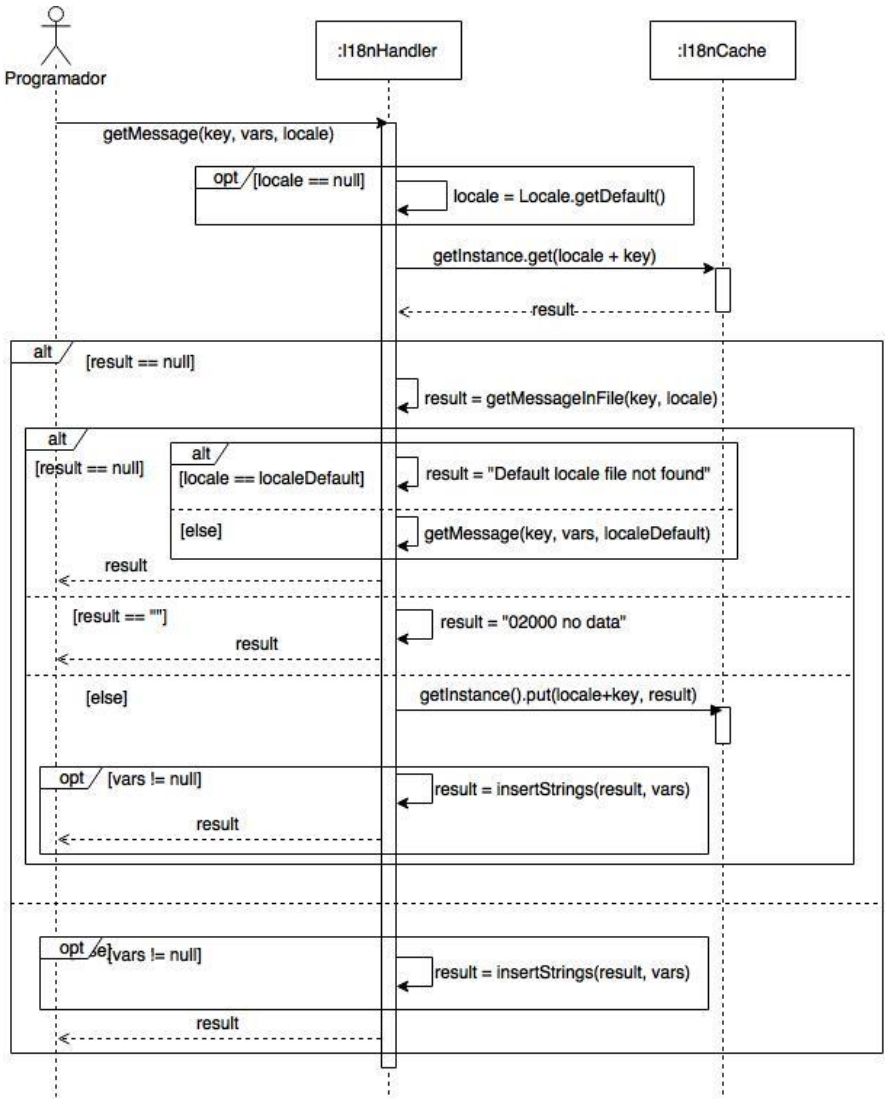


Figura 59 - Consulta de mensagens pelo manipulador I18n – Diagrama de sequência.

É de notar que o método "*getMessage(key, vars, locale)*" é o primeiro método do módulo I18n a ser invocado, quando a instrução "BUNDLE" é executada ao nível do SQL, e que a certa altura existe uma chamada recursiva do mesmo. Esta chamada recursiva acontece quando um ficheiro para um determinado idioma (não padrão) não existe e a pesquisa recomeça de início para o idioma padrão. A mesma só acontece no

máximo uma vez, pois quando o idioma de pesquisa já é o padrão já só devolverá erro ou a mensagem encontrada.

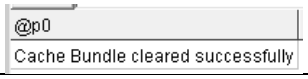
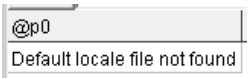
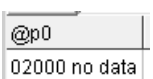
É de salientar que, nesta fase, a lógica aplicada de pesquisa de uma mensagem nos ficheiros externos é inspirada em parte na lógica de pesquisa dos pacotes de recursos da tecnologia Java (Figura 17, Figura 18 e Figura 19).

4.2.4.1 Resultados

Na Tabela 13 são apresentados os resultados dos testes efetuados a esta componente de consulta verificando, assim, o correto funcionamento do algoritmo aplicado. É importante salientar que os testes realizados no ponto 4.2.3.1 complementam, em parte, estes testes de validação da consulta. Para estes testes partiu-se dos seguintes pressupostos:

- O conteúdo dos ficheiros usados nestes testes é o mesmo da Tabela 6.
- O *locale* padrão é en_GB.

Tabela 13 - Testes de consulta do manipulador I18n.

Comando	**Resultado**	**Observação**
call bundle('0001');		Após a base de dados iniciar e ser feito o carregamento incial da *cache*, foi removido o ficheiro *default* da sua diretoria. É devolvida a mensagem que está em *cache*.
call resetbundle('0001');		A mensagem "0001" é eliminada da *cache*.
call bundle('0001');		Não existindo a mensagem na *cache* este vai consultar o ficheiro, que neste momento não existe, logo retorna o erro.
call bundle('0001');		Colocando os ficheiros novamente na diretoria, a mensagem é carregada e devolvida com sucesso.
call bundle('0001');		Removendo novamente os ficheiros da pasta a mensagem permanece carregada na *cache* e é devolvida com sucesso.
call bundle('0005');		Voltando a colocar os ficheiros na diretoria e tentando agora obter uma mensagem que não existe nem na *cache* nem no ficheiro, o respetivo

Comando	Resultado	Observação
		erro é devolvido.
call bundle('0005');	@p0 New test message.	Ao adicionar uma nova mensagem (*"New test message."*) ao ficheiro *default* para a chave "0005" já é possível obter a mesma como retorno.
call bundle('0005', '', 'es_ES');	@p0 New test message.	Como não existe ficheiro para o idioma espanhol de Espanha, a mensagem devolvida foi a do ficheiro *default* (en_GB).
call bundle('0005', '', 'es_ES');	@p0 Nueva mensaje de prueba.	Criando um novo ficheiro "custom-messages_es_ES.txt" na diretoria devidamente preenchido, já foi possível obter a mensagem esperada.
call bundle('0005', '', 'pt_PT');	@p0 02000 no data	Em português de Portugal ainda não existe a mensagem "0005".
call bundle('0005', '', 'pt_PT');	@p0 Nova mensagem de teste.	Adicionando a mesma para a chave "0005" pode-se obter o resultado esperado.

Como se pode constatar, neste momento é possível manipular qualquer mensagem, estando a mesma disponível de imediato, sem que para isso seja necessário interromper a execução da respetiva base de dados.

Por fim, neste ponto já se chegou ao modelo final aplicado a uma base de dados *open source*, tal como mostra a Figura 60.

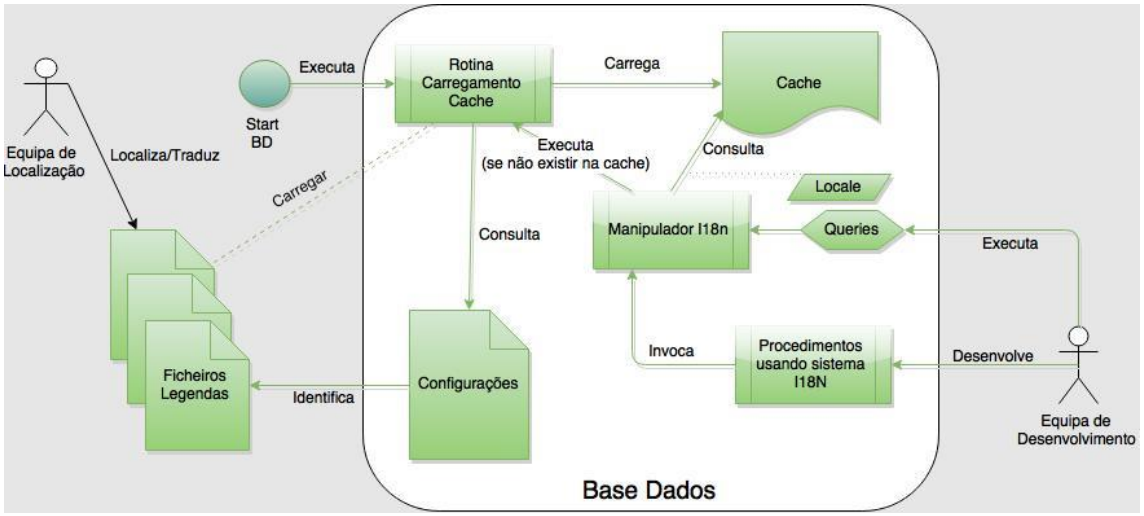


Figura 60 - Quarto e último passo no modelo I18n em base de dados.

4.2.5 Concretização com aplicação Web

Foi simulado um caso de uso deste novo módulo da base de dados com o desenvolvimento de uma pequena e muito simples aplicação web. Esta aplicação consiste apenas em três ecrãs distintos, sendo estes os seguintes:

1. Ecrã inicial onde é possível escolher na *combobox* entre uma mensagem de sucesso ou de erro, clicando posteriormente no botão "*Submit Query*". A ação deste botão é, consoante o tipo de mensagem escolhido, executar um procedimento de base de dados o qual insere, com base no *locale* enviado pelo *browser*, a respetiva mensagem numa tabela criada previamente, denominada por "MSG_RESULT".

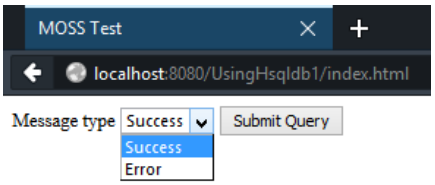


Figura 61 - Primeiro ecrã da aplicação web.

2. Segundo ecrã onde se encontra um *link* que redireciona para o terceiro ecrã.

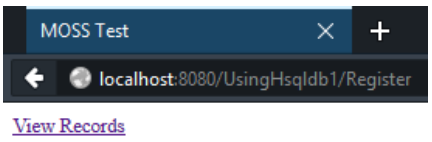


Figura 62 - Segundo ecrã da aplicação web.

3. Terceiro ecrã onde são listados todos os registos da tabela "MSG_RESULT" e onde se encontra um *link* de redireccionamento para o ecrã inicial.

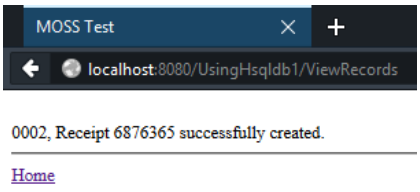


Figura 63 - Terceiro ecrã da aplicação web.

Passando a detalhar o que está por de trás desta pequena aplicação, ao nível da base de dados foi criada uma tabela, para guardar as mensagens solicitadas, com a estrutura apresentada na Figura 64.

```
CREATE TABLE msg_result (code VARCHAR(10), text VARCHAR(255));
```

Figura 64 - Tabela para guardar as mensagens solicitadas.

Foi criado, também, um procedimento de tratamento dos pedidos da aplicação web de acordo com a Figura 65.

```
CREATE PROCEDURE procedure_test(locale VARCHAR(10), type_msg INTEGER)
MODIFIES SQL DATA
BEGIN ATOMIC
      DECLARE msg VARCHAR(255);
      IF(type_msg = 1) -- Success message
      THEN
            SET msg = BUNDLE('0002', '6876365', locale);
            INSERT INTO msg_result (code, text) VALUES ('0002', msg);
      ELSEIF (type_msg = 2) -- Error message
      THEN
            SET msg = BUNDLE('0004', '6876365, 3', locale);
            INSERT INTO msg_result (code, text) VALUES ('0004', msg);
      END IF;
END;
```

Figura 65 - Procedimento de teste na base de dados HSQLDB, o qual usa a nova funcionalidade "BUNDLE".

Se o pedido for de uma mensagem do tipo 1 é feita a inserção na tabela "MSG_RESULT" da mensagem de sucesso "0002". Caso seja de tipo 2, é feita a inserção na mesma tabela da mensagem de erro "0004". É de notar que é neste momento de obtenção da mensagem onde se tem já o suporte da nova instrução "BUNDLE".

No que toca à vertente web, ao submeter o formulário do primeiro ecrã, é invocado o procedimento *"procedure_test"* com o *locale* do pedido, neste caso corresponde ao idioma do *browser* como primeiro parâmetro e o tipo de mensagem selecionada como segundo parâmetro.

```
protected void doPost(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      int typeMsg = Integer.parseInt(request.getParameter("typeMsg"));
      String locale = request.getLocale().toString();
      try {
            PreparedStatement pst=con.prepareStatement("CALL
                                                  procedure_test(?,?)");
            pst.clearParameters();
            pst.setString(1, locale);
            pst.setInt(2, typeMsg);
            pst.executeUpdate();
            out.write("<html>
                        <head><title>MOSS Test</title></head>
                        <body><a href='ViewRecords'>View Records</a></body>
                        </html>");
      } catch (SQLException e) {
            e.printStackTrace(System.out);
      }
}
```

Figura 66 - Método para invocar o procedimento de teste.

Após a execução com sucesso deste procedimento é possível visualizar a totalidade das mensagens já inseridas.

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
                     throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out=response.getWriter();
      try {
            CallableStatement pst=con.prepareCall("Select *
                                                  from msg_result");
            pst.clearParameters();
            ResultSet rs=pst.executeQuery();
            out.write("<html><head><title>MOSS Test</title></head><body>");
            while(rs.next()){
                  out.write("<br/>"+rs.getString(1));
                  out.write(", "+rs.getString(2));
            }
            out.write("<hr/><a href='index.html'>Home</a></html> ");
      } catch (SQLException e) {
            e.printStackTrace(System.out);
      }
}
```

Figura 67 - Método para listar todos os registos da tabela "MSG_RESULT" no terceiro ecrã.

Em qualquer um destes momentos a conexão usada é a seguinte:

- Url: jdbc:hsqldb:http://localhost:80/moss_test
- Utilizador: SA
- Palavra-passe:

```
con=DriverManager.getConnection("jdbc:hsqldb:http://localhost:80/moss_test",
                                "SA","");
```

Figura 68 - Conexão à base de dados HSQLDB de teste.

Antes de executar a aplicação é necessário, primeiro, iniciar a base de dados em modo de servidor web com o seguinte *batch script*:

run_server_hsqlfb.bat

```
1  java -classpath ./hsqldb.jar org.hsqldb.server.WebServer -database.0 moss_test -dbname.0 moss_test
2  pause
```

Figura 69 - Batch Script para iniciar base de dados HSQLDB.

Como se pode verificar na Figura 69, é invocada classe *"WebServer"*, a qual é responsável por iniciar a base de dados em modo servidor web.

Partindo do princípio que já foi gerado um novo ficheiro "*hsqldb.jar"*, que diz respeito ao executável da base de dados em estudo, com base na nova versão do código fonte da mesma, obtém-se a seguinte janela DOS com o seu log de execução.

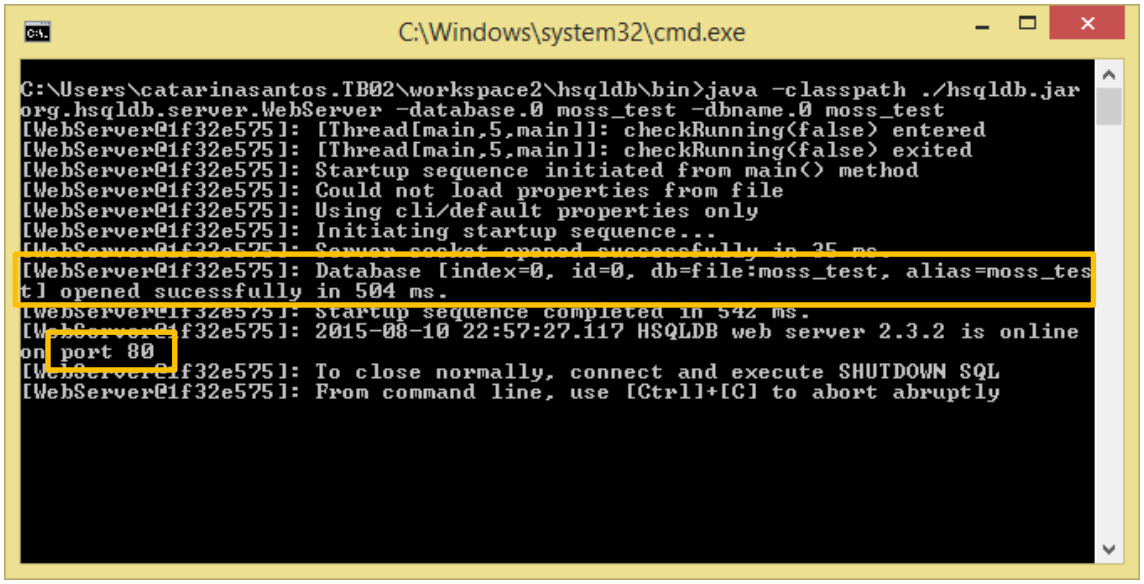


Figura 70 - Log de execução da base de dados HSQLDB (Modo Web Server).

4.2.5.1 Resultados

Na execução dos seguintes testes foram usados dois *browsers* distintos configurados para idiomas diferentes. Os *browsers* usados foram o Chrome (pt_PT - Português de Portugal) e o Firefox (en_GB - Inglês da Grã Bretanha) e ambos executaram a aplicação web desenvolvida.

Partindo do princípio que existem ficheiros de mensagens em inglês, português, francês e italiano, na Tabela 14 são apresentados os resultados obtidos nos testes realizados.

Tabela 14 – Resultados (código, mensagem) dos pedidos de mensagens (sucesso e/ou erro) efetuados pelos browsers.

Chrome	Firefox	Resultado	Observação
Sucesso		0002, O recibo 6876365 foi criado com sucesso.	
	Sucesso	0002, O recibo 6876365 foi criado com sucesso. 0002, Receipt 6876365 successfully created.	
Erro		0002, O recibo 6876365 foi criado com sucesso. 0002, Receipt 6876365 successfully created. 0004, O recibo 6876365 tem 3 itens inválidos.	
	Erro	0002, O recibo 6876365 foi criado com sucesso. 0002, Receipt 6876365 successfully created. 0004, O recibo 6876365 tem 3 itens inválidos. 0004, The receipt 6876365 has 3 invalid items.	
Sucesso		0002, O recibo 6876365 foi criado com sucesso. 0002, Receipt 6876365 successfully created. 0004, O recibo 6876365 tem 3 itens inválidos. 0004, The receipt 6876365 has 3 invalid items. 0002, Reçu 6876365 été créé avec succès.	Alterar idioma do respetivo *browser* para francês de França (fr_FR), antes de efetuar o pedido.

Chrome	Firefox	Resultado	Observação
	Sucesso	0002, O recibo 6876365 foi criado com sucesso. 0002, Receipt 6876365 successfully created. 0004, O recibo 6876365 tem 3 itens inválidos. 0004, The receipt 6876365 has 3 invalid items. 0002, Reçu 6876365 été créé avec succès. 0002, La ricevuta 6876365 è stato creato con successo.	Alterar idioma do respetivo *browser* para italiano (it), antes de efetuar o pedido.
Erro		0002, O recibo 6876365 foi criado com sucesso. 0002, Receipt 6876365 successfully created. 0004, O recibo 6876365 tem 3 itens inválidos. 0004, The receipt 6876365 has 3 invalid items. 0002, Reçu 6876365 été créé avec succès. 0002, La ricevuta 6876365 è stato creato con successo. 0004, Reçu 6876365 dispose de 3 éléments non valides.	
	Erro	0002, O recibo 6876365 foi criado com sucesso. 0002, Receipt 6876365 successfully created. 0004, O recibo 6876365 tem 3 itens inválidos. 0004, The receipt 6876365 has 3 invalid items. 0002, Reçu 6876365 été créé avec succès. 0002, La ricevuta 6876365 è stato creato con successo. 0004, Reçu 6876365 dispose de 3 éléments non valides. 0004, La ricevuta 6876365 ha 3 articoli errati.	

Como se pode constatar nos resultados da tabela acima, a cada pedido efetuado pelos *browsers* é inserida a respetiva mensagem na tabela de acordo com a linguagem atual de cada um deles.

4.2.6 Integração com base de dados fechada

Tendo em conta que a criação deste novo módulo implicou desenvolvimentos ao nível do código fonte da base de dados *open source*, foi estudada a possibilidade de integração do módulo I18n desenvolvido em bases de dados fechadas, isto é, que não permitem alterações ao seu código fonte. A base de dados escolhida para o efeito foi Oracle DB.

Tal como já foi referido anteriormente, o módulo I18n desenvolvido centraliza-se em duas classes, “I18nHandler.java” e “I18nCache.java”. Desta forma, com auxílio da funcionalidade *loadjava* disponibilizada pela base de dados Oracle, foi possível carregar estas duas classes na respetiva base de dados podendo, assim, usufruir das potencialidades de módulo.

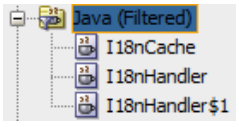


Figura 71 - Classes Java carregadas em Oracle DB.

Uma vez carregadas ambas as classes, previamente compiladas (extensão "*.class"), foi criada a seguinte função:

```
CREATE OR REPLACE FUNCTION bundle(key varchar2,
                                  vars varchar2,
                                  locale varchar2) RETURN STRING AS
LANGUAGE JAVA NAME 'I18nHandler.getMessage (java.lang.String,
                                            java.lang.String,
                                            java.lang.String)
                                            return java.lang.String';
```

Figura 72 - Função "BUNDLE" suportada pela classe Java, "I18nHandler".

Com a criação desta função tem-se a funcionalidade principal de obtenção de uma mensagem específica. Nesta função são sempre passados três argumentos, podendo os dois últimos ser "*null*".

De seguida, foi criado o seguinte procedimento para limpar a *cache*:

```
CREATE OR REPLACE PROCEDURE resetbundle(key varchar2) AS
LANGUAGE JAVA NAME 'I18nHandler.resetCache (java.lang.String)';
```

Figura 73 - Procedimento "RESETBUNDLE" suportado pela classe Java, "I18nHandler".

Com o procedimento apresentado na Figura 73, é possível remover mensagens da *cache*. O argumento de entrada pode ser "*null*" limpando, assim, a cache por completo ou um código de uma mensagem específica.

Segue-se o procedimento para inicialização da *cache*.

```
CREATE OR REPLACE PROCEDURE initBundleCache AS
LANGUAGE JAVA NAME 'I18nHandler.initCache ()';
```

Figura 74 - Procedimento "INITBUNDLECACHE" suportado pela classe Java, "I18nHandler".

O procedimento acima é responsável por efetuar o carregamento inicial das mensagens na *cache*.

Por fim, seguem-se os procedimentos para alteração das variáveis referentes ao módulo I18n.

```
CREATE OR REPLACE PROCEDURE set_bundle_directory(key varchar2) AS
LANGUAGE JAVA NAME 'I18nHandler.setMsgPropsDir (java.lang.String)';
```

Figura 75 - Procedimento "SET_BUNDLE_DIRECTORY" suportado pela classe Java, "I18nHandler".

```
CREATE OR REPLACE PROCEDURE set_bundle_file_default_locale(key varchar2) AS
LANGUAGE JAVA NAME 'I18nHandler.setLocaleDefault (java.lang.String)';
```

Figura 76 - Procedimento "SET_BUNDLE_FILE_DEFAULT_LOCALE" suportado pela classe Java, "I18nHandler".

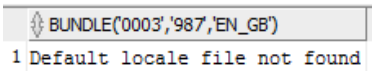
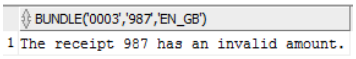
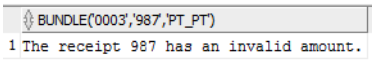
O procedimento da Figura 75 é responsável por alterar a diretoria onde se encontram os ficheiros externos de mensagens. Por outro lado, o procedimento da Figura 76 é responsável por redefinir o idioma referente ao ficheiro padrão de mensagens.

4.2.6.1 Resultados

Foram realizados alguns testes para validar se o comportamento se mantem de acordo com a base de dados HSQLDB, conforme se pode verificar na Tabela 15, com auxilio de um dos ficheiros de *log* da base de dados Oracle. Este ficheiro de *log*, de nome "moss_s000_2552.trc", encontra-se localizado em "\diag\rdbms\moss\moss\trace" dentro da diretoria de instalação da base de dados Oracle. Nele pode-se consultar as mensagens de *log* espalhadas pela classe I18nHandler no momento de consulta das mensagens na *cache* e/ou ficheiro externo. Para os seguintes testes, segue-se do pressuposto que a localização inicial dos ficheiros externos é a pasta *root* da instalação da base de dados Oracle "\product\11.2.0\dbhome_1".

Tabela 15 - Resultados de testes ao módulo I18n em Oracle DB.

Comando	**Resultado**	**Observação**
SELECT bundle('0001', null, null) FROM DUAL;	BUNDLE('0001',NULL,NULL) 1 Bem-vindo! Log: *** 2015-08-28 14:59:23.511 Get Cache init:1317884279784000 Get Cache end:1317884288246000 Get file init:1317884288361000 Get file end:1317884306727000	Mensagem não existe na *cache* na primeira execução.
SELECT bundle('0001', null, null) FROM DUAL;	BUNDLE('0001',NULL,NULL) 1 Bem-vindo! Log: *** 2015-08-28 15:00:50.681 Get Cache init:1317971451091000 Get Cache end:1317971452238000	Mensagem já se encontra carregada na *cache*.
SELECT bundle('0003', '987', null) FROM DUAL;	BUNDLE('0003','987',NULL) 1 O recibo 987 tem um montante inválido.	Substituição de variáveis.
SELECT bundle('0003', '987', 'en_GB') FROM DUAL;	BUNDLE('0003','987','EN_GB') 1 The receipt 987 has an invalid amount.	Indicação de idioma específico, neste caso "en_GB".

Comando	Resultado	Observação
EXECUTE resetBUNDLE(null);		Limpar *cache*.
EXECUTE set_bundle_directory('C:\\');		Mudar localização dos ficheiros para "C:\".
SELECT bundle('0003', '987', 'en_GB') FROM DUAL;		Sem ficheiros externos na diretoria "C:\".
SELECT bundle('0003', '987', 'en_GB') FROM DUAL;	 Log: *** 2015-08-28 21:16:44.056 Get Cache init:1340531800782000 Get Cache end:1340531802324000 Get file init:1340531802436000 Get file end:1340531812023000	Com ficheiros externos na diretoria "C:\".
EXECUTE SET_BUNDLE_FILE_DEFAULT_LOCALE('pt_PT');		Definir o *locale* do ficheiro padrão como "pt_PT".
EXECUTE resetBUNDLE(null);		Limpar *cache*.
SELECT bundle('0003', '987', 'pt_PT') FROM DUAL;		Com *locale* "pt_PT" devolve mensagem do ficheiro padrão.

Com os testes realizados acima verificou-se que o principal objetivo foi atingido, que era:

- Consultar mensagem na *cache.*
- Consultar mensagem no ficheiro externo, caso esta não exista na *cache*.
- Ao encontrar a mensagem no ficheiro, carregá-la na *cache* e devolve-la com sucesso.
- Poder alterar localização dos ficheiros externos e idioma do ficheiro padrão.

No entanto, verificou-se que, por se tratar de classes Java carregadas em base de dados, as mensagens guardadas em *cache* e as alterações de diretoria ou idioma de ficheiros permanecem apenas por sessão, isto é, a cada nova sessão existe uma nova *cache* I18n

encontrando-se a mesma vazia e as variáveis de diretoria e idioma com os valores padrão. Correlacionado com este tema está o facto de, para já, não se conseguir efetuar o carregamento inicial da *cache* automaticamente, efetuando-o apenas manualmente com o procedimento criado “INITBUNDLECACHE”. Neste sentido, há algum trabalho a ser desenvolvido. Ainda assim, este pode não ser um problema se se garantir que uma aplicação usa sempre a mesma conexão à base de dados, pois a *cache* em uso será sempre a mesma para aquela conexão.

5 Conclusões e Trabalhos Futuros

5.1 Conclusões

Esta dissertação abordou o tema de localização de *software* em bases de dados *open source* e provou que é possível incluir no seu *core* um modelo de internacionalização suficientemente simples e flexível tanto para o programador como para o tradutor, não interferindo na disponibilização da mesma.

Nesta dissertação foi feita alguma revisão de literatura acerca de localização de *software* em geral atingindo, assim, o primeiro objetivo o qual consiste na identificação do estado da arte. Com o auxílio da revisão de literatura concretizou-se o segundo objetivo, sendo definido o modelo conceptual i18n a ser desenvolvido. Tendo o modelo definido, o mesmo foi aplicado a uma base de dados *open source* específica, nomeadamente a HSQLDB. A concretização do modelo foi efetuada com sucesso ficando, assim, o terceiro objetivo concluído.

Por último, foi validada a possibilidade de integração do novo módulo desenvolvido em Java numa base de dados fechada, nomeadamente Oracle DB. Esta integração, respeitante ao quarto e último objetivo, foi validada com sucesso, pois as principais funcionalidades do módulo comportaram-se como era suposto, nomeadamente:

- Consultar uma mensagem na *cache*.
- Consultar uma mensagem nos ficheiros externos, caso esta não exista na *cache*.
- Carregar na *cache* e devolver mensagem, após encontrá-la no ficheiro externo.
- Substituir variáveis nas mensagens.
- Alterar propriedades de localização dos ficheiros externos e idioma do ficheiro padrão.

Após esta validação verificou-se que a informação em *cache* permanece apenas por sessão, isto é, entre sessões de base de dados diferentes existe novas caches de internacionalização. Este acaba por não ser um problema se se garantir que uma aplicação, durante a sua execução, usa sempre a mesma conexão à base de dados e, assim, a sessão e *cache* em uso será sempre a mesma.

Em suma, pode-se apontar como principal contribuição deste trabalho, a implementação de uma *framework* I18n capaz de habilitar um motor de base de dados a desenvolver

bases de dados facilmente internacionalizáveis. O modelo conceptual da mesma foi aplicado com sucesso numa base de dados *open source* e a integração do mesmo em base de dados fechada foi validada igualmente com sucesso.

5.2 Trabalhos Futuros

De forma a complementar esta dissertação e como sugestão de trabalhos futuros, pode-se enumerar:

- Investigar integração do modelo criado em bases de dados fechadas

  Como já foi referido anteriormente, não houve oportunidade para uma investigação mais exaustiva, acompanhada de testes, no que toca à integração do novo modelo em bases de dados fechadas, uma vez que não é permitido alterar código fonte das mesmas. Por isso, sugere-se uma investigação mais profunda no que toca ao tema de memória/*cache* e sessões em bases de dados fechadas, uma vez que atualmente a cache I18n é mantida apenas por sessão.

- Trabalhar com linguagens de caracteres superiores a um byte (asiáticas)

  É interessante potenciar o módulo desenvolvido no que toca a codificação de caracteres, pois é uma mais-valia este suportar o maior número de linguagens possível.

- Permitir definir nas próprias mensagens a ordem dos valores variáveis

  Neste momento o módulo desenvolvido suporta mensagens com elementos variáveis podendo, assim, atribuir valores aos mesmos no momento da sua chamada. No entanto, a substituição destes valores é sequencial conforme a sequência dos valores que se indicar na sua chamada. Seria interessante poder controlar na estrutura da própria mensagem a ordem com que os valores da lista indicada são colocados na mesma. Esta melhoria pode ser bastante útil, uma vez que a estrutura de uma mesma mensagem pode ser completamente diferente, por exemplo, de português ou inglês para alemão.

6 Referências Bibliográficas

Aragão, D. (2004). O sistema UNL nos processos de internacionalização e localização de software. Em Universidade Federal de Santa Catarina, Florianópolis. Obtido de https://projetos.inf.ufsc.br/arquivos_projetos/projeto_308/TCC_Diego.pdf. [Online; acedido em 22-Maio-2015]

Azenha, B. (2006). Armazenamento de Informação em HSQL. Em Instituto Superior Técnico, Universidade Técnica Lisboa Taguspark, Portugal. Obtido de https://fenix.tecnico.ulisboa.pt/downloadFile/3779571248003/TDArmazenamentoHSQLFinal.doc. [Online; acedido em 06-Setembro-2015]

Batista, M. (2014). Abordagem metodológica para auxiliar no processo de localização de um ERP Open Source. Em ISCTE-IUL. Obtido de https://repositorio.iscte-iul.pt/handle/10071/8696. [Online; acedido em 22-Maio-2015]

Cordioli, E. (2006). Empreendimento em Vigilância Tecnológica para Localização e Internacionalização de Software. Em Florianópolis. Obtido de https://projetos.inf.ufsc.br/arquivos_projetos/projeto_441/MonografiaEmanuelComFonte.pdf. [Online; acedido em 18-Setembro-2015]

Darcy G. Benoit, T. M. (2004). IDUX: Internationalization of Data Using XML. Em Conference: Proceedings of the IADIS International Conference WWW/Internet 2004, Madrid, Spain. Obtido de http://www.researchgate.net/publication/220969007_IDUX_Internationalization_of_Data_Using_XML. [Online; acedido em 22-Maio-2015]

Deitsch, A., & Czarnecki, D. (2001). Java Internationalization. O'Reilly Media, Inc. Obtido de

https://books.google.pt/books?hl=en&lr=lang_en%7Clang_pt&id=djuMgaEuBhQC&oi=fnd&pg=PR9&dq=resource+bundle+java+architecture&ots=kJoZagD36x&sig=uEVeBfbrKyc93JvHzoNhOe7CgVg&redir_esc=y#v=onepage&q&f=false. [Online; acedido em 02-Setembro-2015]

Gillam, R. (1999). Developing Global Applications in Java. Em 15th International Unicode Conference, San Jose, CA. Obtido de http://unicode.org/iuc/iuc15/ta4/slides.pdf. [Online; acedido em 22-Maio-2015]

Gross, S. (2006). Internationalization and Localization of Software. Em University Eastern Michigan University, Michigan, USA. Obtido de http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.175.2883&rep=rep1&type=pdf. [Online; acedido em 22-Maio-2015]

Gut, A., Miclea, L., Enyedi, S., Abrudean, M., & Hoka, I. (2006). Database Globalization in Enterprise Applications. Em 2006 IEEE International Conference on Automation, Quality and Testing, Robotics. Obtido de http://doi.org/10.1109/AQTR.2006.254560. [Online; acedido em 22-Maio-2015]

Hau, E., & Aparício, M. (2008). Software Internationalization and Localization in Web Based ERP. Apresentado na SIGDOC ’08, New York. Obtido de https://vpn2.iscte.pt/+CSCO+0h756767633A2F2F71792E6E707A2E626574++/citation.cfm?doid=1456536.1456570. [Online; acedido em 26-Setembro-2015]

Jellema, L. (2012). Supporting multiple languages in ADF applications backed by resource bundles - and programmatically controlling the JSF locale. Obtido de https://technology.amis.nl/2012/08/11/supporting-multiple-languages-in-adf-applications-backed-by-resource-bundles-and-programmatically-controlling-the-jsf-locale/. [Online; acedido em 13-Setembro-2015]

Jenkov, J. (2014). Java ResourceBundle. Obtido de http://tutorials.jenkov.com/java-internationalization/resourcebundle.html. [Online; acedido em 20-Setembro-2015]

Kozovits, L., Melo, R., & Feijó, B. (2003). Um estudo do uso de SGBDs relacionais em arquiteturas de jogos multi-jogador. Em PUC-Rio. Obtido de http://www.dbd.puc-rio.br/depto_informatica/03_35_kozovits.pdf. [Online; acedido em 01-Setembro-2015]

Lozano, F. (2008). Artigo Java Magazine 30 - O Novo HSQLDB. Obtido de http://www.devmedia.com.br/artigo-java-magazine-30-o-novo-hsqldb/8757. [Online; acedido em 18-Junho-2015]

Mantoan, F. (2012). Internacionalização e Localização com Zend Framework 1. Obtido de http://blog.fernandomantoan.com/internacionalizacao-e-localizacao-com-zend-framework/. [Online; acedido em 13-Setembro-2015]

McCollough, M. (2011). Software Design Considerations for Internationalization and Localization | bridge360blog [Blog]. Obtido de http://bridge360blog.com/2011/11/25/software-design-considerations-for-internationalization-and-localization/. [Online; acedido em 22-Maio-2015]

McEnery, A. M., & Xiao, R. Z. (2005). Character encoding in corpus construction. Em M. Wynne (Ed.), Developing Linguistic Corpora : A Guide to Good Practice. Oxford, UK: AHDS. Obtido de http://eprints.lancs.ac.uk/60/. [Online; acedido em 22-Maio-2015]

Oracle. (2014). Internationalization Overview. Obtido de http://docs.oracle.com/javase/7/docs/technotes/guides/intl/overview.html. [Online; acedido em 19-Setembro-2015]

Prudêncio, A. C., Valois, D. A., & Lucca, J. E. D. (2004). Introdução à internacionalização e à localização de softwares. Em Cadernos de Tradução. Obtido de https://periodicos.ufsc.br/index.php/traducao/article/view/6482. [Online; acedido em 22-Maio-2015]

Resende, H., & Silva, M. M. da. (2009). Gestão de projectos de tradução e de localização - do conceito ao modelo. Obtido de http://recipp.ipp.pt/handle/10400.22/2883. [Online; acedido em 22-Maio-2015]

Ribeiro, G. C. B. (2005). Tradução e localização de software e outros produtos: Audiovisual ou Multimídia? Em Cadernos de Tradução. Obtido de https://periodicos.ufsc.br/index.php/traducao/article/view/6742. [Online; acedido em 22-Maio-2015]

Wave, R. (2011). Localized Resources. Obtido de http://docs.roguewave.com/sourcepro/12.0/html/i18nug/10-4.html. [Online; acedido em 17-Setembro-2015]

Xoriant. (2002). Internationalization Challenges and Solutions. Obtido de http://www.xoriant.com/white-paper/internationalization-challenges-and-solutions. [Online; acedido em 17-Setembro-2015]

Zend. (2015a). Overview - Introduction to Zend Framework - Zend Framework. Obtido de http://framework.zend.com/manual/1.12/en/introduction.overview.html. [Online; acedido em 17-Setembro-2015]

Zend, F. (2015b). Zend_Translate - Zend Framework. Obtido de http://framework.zend.com/manual/1.12/en/zend.translate.html. [Online; acedido em 13-Setembro-2015]