UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

CURSO DE CIÊNCIA DA COMPUTAÇÃO

FELIPE ROOS DA ROSA

Promovendo o Reuso de Software através da RAS

Trabalho de Graduação.

Prof. Dr. Marcelo Soares Pimenta

Orientador

Porto Alegre, janeiro de 2009.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitora de Graduação: Profa. Valquiria Link Bassani

Diretor do Instituto de Informática: Prof. Flávio Rech Wagner

Coordenador do CIC: Prof. João César Netto

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Agradecimentos

Quando desejado pelo autor do trabalho, são apresentados logo após a folha de rosto, nessa ordem. São de livre apresentação gráfica. Geralmente os Agradecimentos são apresentados como um capítulo não-numerado. O mesmo é recomendado para os próximos itens, até o Resumo e Abstract.

Sumário

LISTA DE ABREVIATURAS E SIGLAS 6

LISTA DE FIGURAS 7

LISTA DE TABELAS 8

RESUMO 9

1 Introdução 11

2 Adoção do Reuso de Software na atualidade 13

2.1 Os Desafios na Adoção do Reuso de Software 13

2.2 A Importância do Reuso de Software 14

3 Repositórios de Software 15

3.1 Repositório de Gerência de Configuração 15

3.1.1 Características Chaves 15

3.1.2 Aplicação Prática 16

3.2 Repositório de Reuso 16

3.2.1 Características Chaves 16

3.2.2 Aplicação Prática 16

3.3 Ciclo de Vida dos Artefatos de Software 16

3.4 Ferramentas Existentes 17

3.4.1 Gerência de Configuração 17

3.4.2 Reuso 17

3.5 Ferramentas que Suportam o RAS 17

3.5.1 Basic Asset Retrieval Tool e Component Repository 17

3.5.2 Reusable Asset Manager 17

3.5.3 ArcSeeker 17

3.5.4 Outros!!!! 18

4 Promovendo o Reuso de Software utilizando RAS 19

4.1 Relação entre Geração e Consumo de Artefatos 19

4.2 O artefato RAS 20

4.3 Suporte do formato RAS no Archiva 22

4.3.1 Consumidores Archiva 22

4.3.2 O Arquivo POM 23

4.3.3 Colocando artefatos RAS no Archiva 24

4.3.4 Adaptação para Apresentação dos Resultados 29

4.3.5 Recuperação e Pesquisa de Artefatos no Archiva 31

5 Conclusão 34

glossário 35

anexo A <Descrição do anexo> 36

anexo b <EXEMPLO dE anexo> 37

apêndice Esquema XSD Para Descritor de ativo de Repositório 39

LISTA DE ABREVIATURAS E SIGLAS

BB Banco do Brasil

CC Código Civil

BR Brasil

UFRGS Universidade Federal do Rio Grande do Sul

BD Banco de Dados

LISTA DE FIGURAS

Figura 4.1 – Relação dos elementos do domínio 19

Figura 7.1: Exemplo de apresentação de uma figura no texto (MEREGALI, 2004). **Error! Bookmark not defined.**

Figura 7.2: Outro exemplo de figura **Error! Bookmark not defined.**

Figura 7.3: Terceira figura: Local das férias depois do Trabalho de Conclusão **Error! Bookmark not defined.**

LISTA DE TABELAS

Tabela 6.1: Parâmetros para formatação das subdivisões do texto **Error! Bookmark not defined.**

Tabela 7.1: Exemplo de apresentação de uma tabela no texto **Error! Bookmark not defined.**

Tabela 8.1: Deve-se escolher somente um tipo de citação para usar durante o texto **Error! Bookmark not defined.**

Tabela 8.2: observação quanto às aspas **Error! Bookmark not defined.**

RESUMO

Consiste na apresentação clara e concisa dos pontos relevantes do trabalho, de maneira a permitir ao leitor saber da conveniência ou não da sua leitura na íntegra. É redigido pelo autor, em português e em inglês, em páginas distintas, antecedendo a introdução. Cada um ocupará no máximo 1 folha, e poderão ter *até 500 palavras*. Para maiores informações com relação à redação consultar a NBR-6028 da ABNT ([1990](http://index.html/#ABNT:NBR-6028-1990)).

Quanto ao estilo, o resumo deve ser composto por uma seqüência de frases completas e não por uma enumeração de tópicos; a primeira frase deverá ser significativa, explicando o tema principal do documento. Na redação, dar preferência ao uso da terceira pessoa do singular e do verbo na voz ativa. Após o resumo e o abstract devem constar palavras-chave relativas aos assuntos da monografia, em português e inglês respectivamente. Estas são alinhadas na margem inferior do documento.

A ABNT define resumo como: “[...] seqüência de frases concisas e objetivas e não de uma simples enumeração de tópicos, não ultrapassando 500 palavras, seguido logo abaixo, das palavras representativas do conteúdo do trabalho, isto é, palavras-chave e/ou descritores, conforme a NBR 6028.”

Este item serve para informar o conteúdo do trabalho, orientando assim, o leitor na certeza da continuidade, ou desistência  da leitura do mesmo.

Palavras-Chave: ABNT, processadores de texto, formatação eletrônica de documentos.

Promoting Software Reuse Through RAS

ABSTRACT

This manual has the purpose of…

O abstract deve apresentar, adicionalmente, uma tradução do título do trabalho. O título traduzido é colocado antes do título do capítulo (“Abstract’”), a 2cm da margem superior, centralizado, em fonte Times 14 pt negrito.

Ocupará no máximo 1 folha, e poderão ter *até 500 palavras*. Para maiores informações com relação à redação consultar a NBR-6028 da ABNT ([1990](http://index.html/#ABNT:NBR-6028-1990)).

Trabalhos de conclusão de especialização normalmente não exigem a inclusão de Abstract em inglês.

**Keywords:** ABNT, text processors, electronic document preparation.

# Introdução

A idéia de reutilizarmos software é uma técnica que tem sido estudada por décadas, desde os primórdios da Engenharia de Software (McIlroy, 1968). O objetivo final era bem claro: construir alguma coisa apenas uma vez e reulizá-la diversas vezes. Desta maneira, o custo seria reduzido, porque o tempo que seria necessário para repetir uma atividade poderia ser investido em outras tarefas relevantes.

Embora o reuso de software seja uma técnica cuja discussão é extensiva, muitas razões existem para que ela não seja praticada. A grande maioria delas pode ser classificada como psicológica, sociológica ou econômica. A única razão técnica é a falta de métodos de pesquisa para encontrar as peças necessárias, ou a qualidade pobre dos componentes . Dentro deste contexto, Sherif coloca também que a maior barreira técnica está na dificuldade em localizar artefatos reusáveis .

Um repositório de reuso é uma coleção de artefatos reusáveis com requisitos tais como mecanismos de recuperação e pesquisa . Esta ferramenta se propõe a resolver o problema técnico da localização de artefatos. Apesar disso, a literatura existente diverge quanto à importância da ferramenta na promoção e adoção do reuso. Frakes e Fox revelam que repositórios de reuso são importantes, mas não caracterizam um fator decisivo na adoção do reuso . Contudo, Sherif propõe que a falta de repositórios de reuso compõe uma barreira para adoção do mesmo . Apesar disso, Lucresio constata que a existência de repositório de reuso não contribui para o sucesso do reuso de software .

Em parte, o problema de encontrar artefatos reusáveis está também na indefinição quanto a descrição necessária. Ezran coloca que material reusável deve ser empacotado juntamente com toda a informação necessária para o seu reuso . A OMG resolveu este problema quando do estabelecimento da RAS (Reusable Asset Specification), uma especificação para descrição e empacotamento de bens reusáveis.

Neste sentido, é visível a necessidade de integrarmos a solução para a descrição e classificação dos artefatos com um repositório de reuso. O fato de a RAS ser uma especificação pode colaborar para que ela seja adotada em larga escala. Entretanto, para que isto ocorra, é necessário que existam soluções que suportem o seu formato. Sendo a RAS, na escrita deste documento, um padrão recente, existe uma carência de ferramentas que a suportem. Atualmente, a lista de aplicações disponíveis que são compatíveis com este formato é restrita ou limitada a soluções proprietárias. Assim, é interessante que uma solução possa ser desenvolvida dentro ou a partir de uma solução de código-fonte aberto.

Assim, este trabalho dá continuidade ao iniciado por Martins em , no sentido de torná-lo mais de acordo com os padrões de mercado. Naquele trabalho, o desenvolvimento de uma ferramenta de suporte a reuso é completo, porém auto-contido em suas definições. Com o advento da RAS, podemos modificar isto na esperança de que a especificação vire um padrão de facto. Portanto, este trabalho propõe uma infraestrutura de suporte ao padrão RAS quanto à geração, armazenamento e consumo de artefatos reusáveis. Para tanto, vamos fazer com que um gerenciador de repositórios chamado Archiva (mantido pela comunidade Apache) suporte a definição RAS.

O documento está estruturado como segue. No capítulo 2 apresentamos uma breve discussão sobre a adoção do reuso, retomando algumas idéias referente a sua prática – benefícios e barreiras – reforçando a motivação para este trabalho.

No capítulo 3 abordamos alguns conceitos relativos aos repositórios. Lá fazermos uma revisão de literatura para clarificar os conceitos de repositório de reuso e repositório para gerência de configuração, expressando as diferenças e semelhanças entre eles. Exemplificamos alguns conceitos com exemplos de repositórios de gerência de configuração que existem na atualidade e, também, discutimos algumas alternativas existentes para repositórios de reuso.

No capítulo 4 definimos a estrutura geral e o protótipo do RASPUTIN. A implementação é apresentada em 3 sub-partes relativas ao armazenamento, apresentação e busca, respectivamente.

No capítulo 5 concluímos com os principais achados durante a implementação e idéias para trabalhos futuros.

# Adoção do Reuso de Software na atualidade

Este capítulo desenvolve uma breve discussão sobre a problemática da adoção do reuso de software na atualidade, comparando achados de textos publicados. Primeiramente, abordamos os desafios existentes e barreiras para adoção do reuso de software. Em seguida, destacamos alguns pontos importantes onde o reuso de software pode ajudar e, para finalizar o capítulo, apresentamos a diferença entre o desenvolvimento de software *com* reuso de software e o desenvolvimento de software *para* o reuso de software.

## Os Desafios na Adoção do Reuso de Software

O reuso de software já foi considerado a grande técnica para resolução dos problemas de Engenharia de Software. Entretanto, sua adoção nos dias atuais passa por dificuldades que vão além das limitações técnicas. Diversas barreiras são conhecidas para que o reuso não ocorra. Dentre os problemas descritos, encontram-se fatores gerenciais, organizacionais, econômicos, conceituais ou técnicos .

**Obstáculos Gerenciais e Organizacionais**. Como observamos, reuso não é apenas um problema técnico que deve ser resolvido por engenheiro de software. Assim, suporte gerencial e estruturas organizacionais adequadas são igualmente importantes. Os mais obstáculos mais comuns são:

* **Falta de suporte gerencial**. Já que reuso de software causa custos a curto-prazo, não há como alcançá-lo em uma organização sem suporte da alta gerência. Gerentes devem ser informados sobre o custo inicial e devem ser convencidos sobre as expectativas de redução de custo.
* **Gerenciamento de projeto**. Gerenciar projetos tradicionais não é uma tarefa fácil, principalmente, projetos relacionados com reuso de software. Um passo na direção da adoção de reuso em larga escala tem um impacto em todo o ciclo de vida da produção de software.
* **Estruturas organizacionais inadequadas**. Estruturas organizacionais devem considerar diferentes necessidades que se mostram quando reuso explícito e em larga escala é adotado. Por exemplo. um time separado pode ser definido para desenvolver, manter e certificar componentes de software; e
* **Iniciativas gerenciais**. A falta de iniciativas proíbe gerentes de deixar seus desenvolvedores gastem tempo e construam componentes de um sistema reusável. O sucesso deles é freqüentemente medido apenas no tempo necessário para completar um projeto. Realizar qualquer trabalho além daquele, ainda que benéfico para a companhia como um todo, diminui o seu sucesso. Mesmo quando componentes são reusados acessando repositórios de software, os benefícios obtidos são apenas uma fração do que poderia ser alcançado com reuso explícito, planejado e organizado.

**Obstáculos Econômicos**. Reuso pode economizar dinheiro em longo prazo, mas não sai de graça. Custos associados com o reuso podem ser: custo de construir alguma coisa reusável, custo de reusá-la, e custos de definir e implementar um processo de reuso. Reuso requer investimentos de curto-prazo em infra-estrutura, metodologia, treinamento, ferramentas e arquivos, com resultados sendo realizados apenas anos depois. Desenvolver bens para reuso é mais caro do que desenvolvê-los para um único uso. Níveis mais altos de qualidade, confiabilidade, portabilidade, manutenibilidade, generalidade e documentação mais completa são necessários. Tais custos aumentados são se justificam se o componente é reutilizado apenas uma vez.

**Obstáculos Conceituais e Técnicos**. Os obstáculos técnicos para o reuso de software incluem problemas relacionados a pesquisar e recuperar componentes, componentes legados e aspectos incluindo adaptação.

* **Dificuldade de encontrar software reusável**. Para reusar componentes de software devem existir meios eficientes para encontrar e recuperar os mesmos. Adicionalmente, é importante ter um repositório bem organizado contendo componentes com algum tipo de acesso.
* **Não reusabilidade de software encontrado**. Acesso fácil a software existente não necessariamente promove o reuso de software. Bens reusáveis devem ser cuidadosamente especificados, projetados, implementados e documentados, assim, algumas vezes, modificar e adaptar software pode ser mais custoso do que programar a funcionalidade necessária desde o começo.
* **Componentes legados não preparados para reuso**. Uma abordagem conhecida para reuso de software é o uso de software herdado. Entretanto, simplesmente recuperar os bens existentes de um sistema legado e tentar reusá-los para novos desenvolvimentos não é suficiente para reuso sistemático. Reengenharia pode ajudar na extração de componentes reusáveis de sistemas legados, mas os esforços necessários para entendimento e extração devem ser considerados; e
* **Modificação**. É muito difícil encontrar um componente que funcione exatamente da maneira que se quer. Assim, modificações são necessárias e devem existir maneiras para determinar seus efeitos nos componentes e sua prévia verificação de resultados.

## A Importância do Reuso de Software

O que a bibliografia fala sobre isso?

## Desenvolvendo com Reuso e para Reuso

Como vimos anteriormente, a idéia do reuso está sendo discutida já há bastante tempo. Entretanto, descobriu-se que sua adoção não é tão fácil quando parecia. De fato, percebeu-se que para tirarmos proveito do reuso de software é preciso que ele seja um processo *sistemático*.

# Repositórios de Software

## Repositório de Gerência de Configuração

Texto introdutório

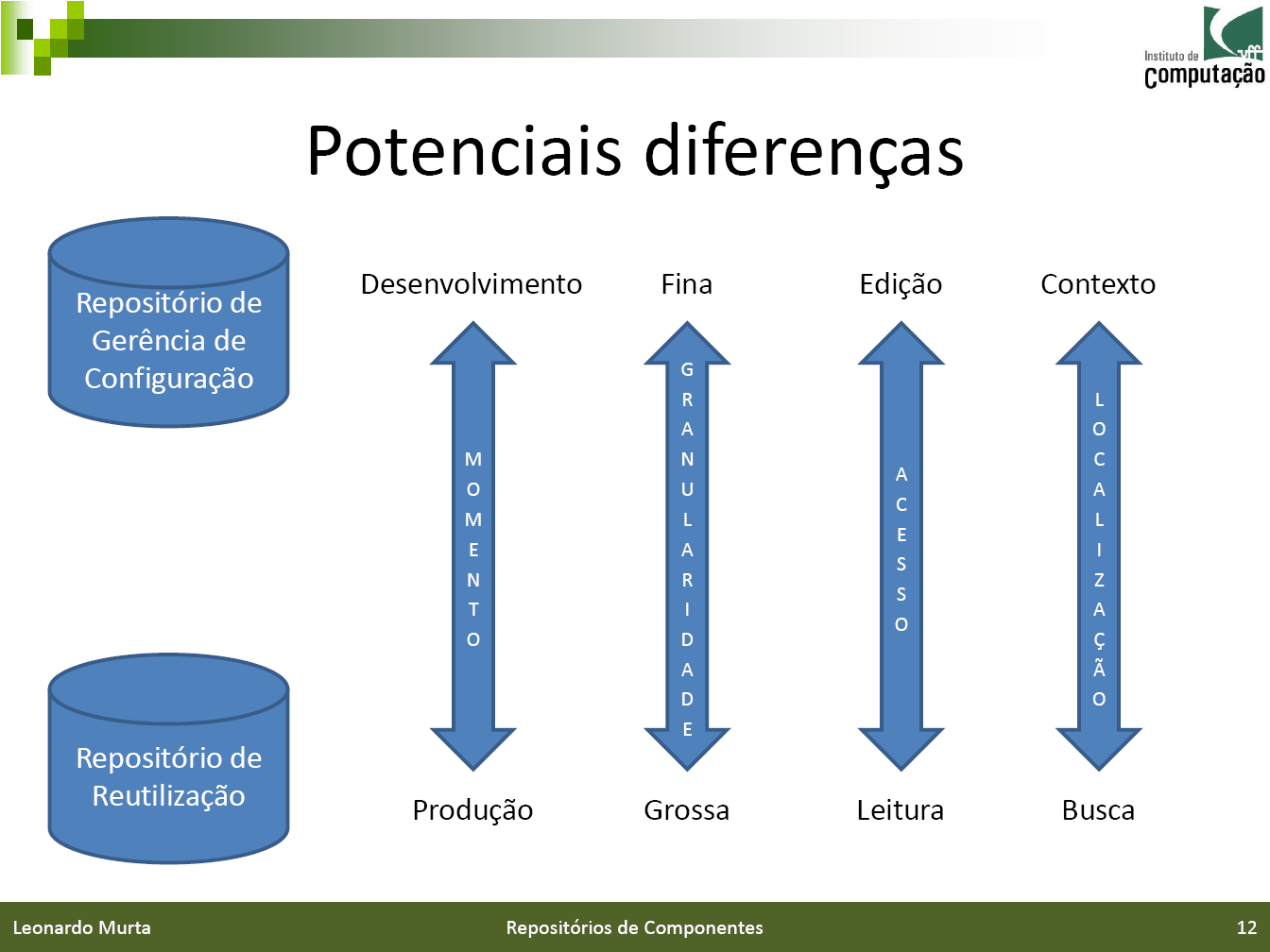


Figura .: Repositório de Gerência de Configuração versus Repositório de Reutilização

### Características Chaves

### Aplicação Prática

## Repositório de Reuso

Texto introdutório

### Características Chaves

### Aplicação Prática

## Ciclo de Vida dos Artefatos de Software

Ciclo de vida dos artefatos de software envolvendo ambos repositórios, desde sua criação, desenvolvimento, publicação, manutenção e republicação. Conecta a visão dos dois repositórios. Onde conseguir o diagrama??? .

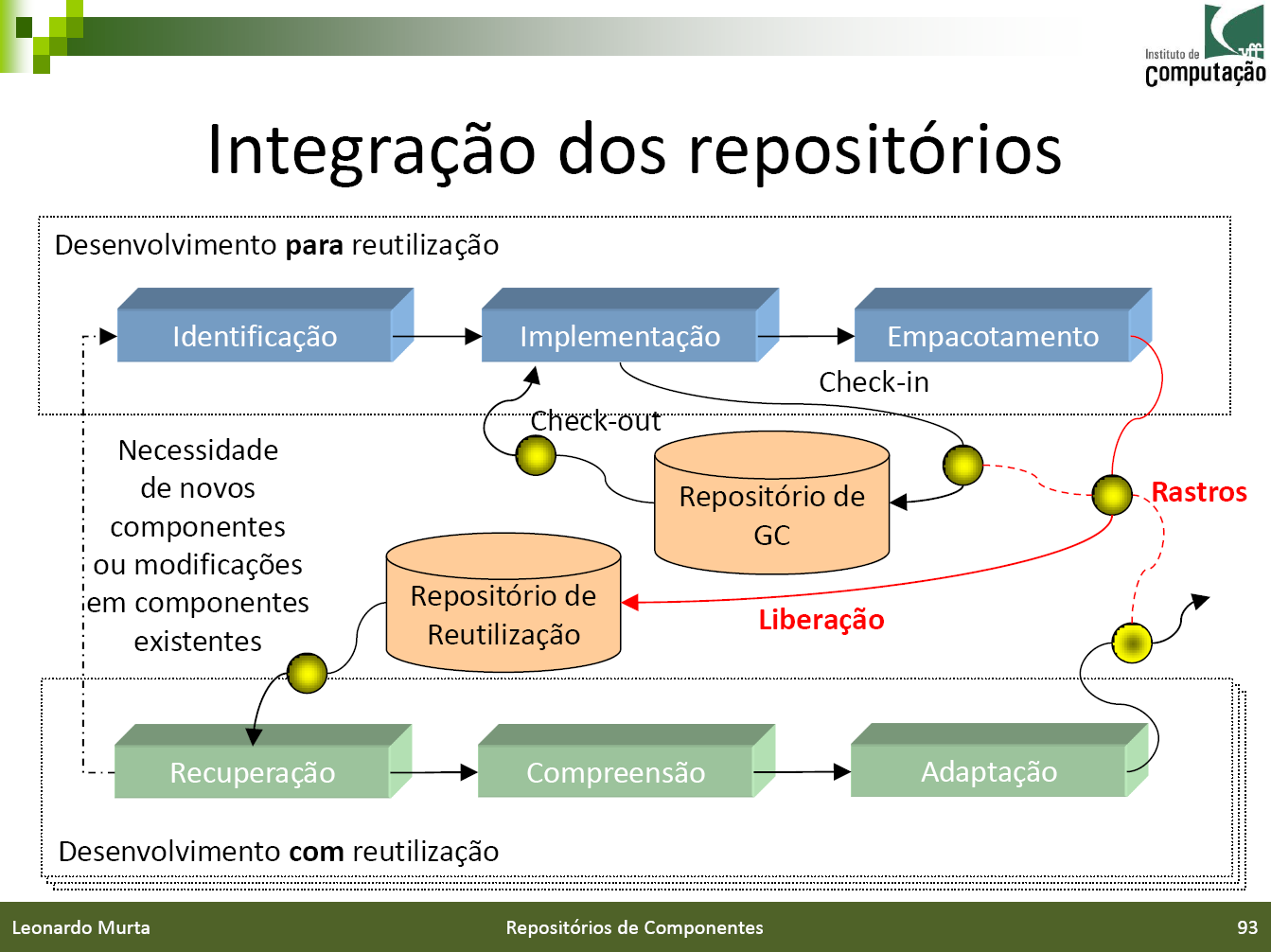


Figura .: Integração dos Repositórios

## Ferramentas Existentes

Embora o conceito seja relativamente fácíl de se assimilar, poucas ferramentas atualmente implementam a especificação RAS. Abaixo listamos alguns exemplos de softwares que oferecem algum tipo de suporte a reuso, mas que não se empenham em suportar o RAS.

### Gerência de Configuração

#### Concurrent Version Systems (CVS)

#### Subversion (SVN)

### Reuso

#### Maven

#### Archiva

#### Nexus

#### SourceForge

#### Google Code

## Ferramentas que Suportam o RAS

### Basic Asset Retrieval Tool e Component Repository

### Reusable Asset Manager

### ArcSeeker

### Outros!!!!

# Promovendo o Reuso de Software utilizando RAS

Podemos observar que um ambiente prático de suporte a reuso implica, apesar das divergências, em termos ferramentas adequadas. Neste contexto, ferramentas como gerenciador de dependências Maven podem ser muito uteis para promover o reuso. A maneira como a ferramenta Maven e o “mundo Java” se interconectam fornecem uma boa visão para a qual um ambiente de reuso deve caminhar.

Observamos também que não são muitas as opções disponíveis em código fonte aberto quando o requisito não-funcional principal se refere ao suporte da especificação RAS. Sabemos que o caminho para o reuso de software não passa necessariamente por esta especificação, dado que o reuso já foi caso de sucesso em diversas empresas antes da origem de tal especificação. Entretanto, dado a seriedade e profissionalismo da organização que a propôs, esperamos que esta definição venha a tornar-se um padrão a ser seguido por todas as instituições defensoras do reuso de software. Daí a importância de compartilharmos desta idéia e darmos um passo em direção a adoção extensiva desta especificação. Este capítulo dedica-se a enfrentar este desafio.

A idéia de toda infra-estrutura de suporte à especificação RAS é fornecer mecanismos pra que alguém possa gerar, armazenar, procurar e recuperar artefatos RAS de uma maneira integrada. Com isso em mente, podemos observar o que é objetivo deste trabalho inicial na Figura 4.1.



Figura .: Infra-estrutura de suporte ao padrão RAS

Como podemos intuir da descrição dos componentes da figura, temos dois ambientes. O primeiro se refere ao ambiente do cliente. Ele representa o ambiente ao qual o desenvolvedor está ligado, seja para *promover* o reuso quanto para *fazer uso* do mesmo. Sendo assim, temos dois sistemas-atores principais: o gerador RAS, responsável por gerar artefatos que sigam a especificação RAS; e o recuperador RAS, que integra o uso e geração de artefatos ao sistema de disponibilização de artefatos RAS (atualmente representado por um navegador web).

O segundo ambiente, foco do RASPUTIN, é onde se encontram os componentes responsáveis por possibilitar o armazenamento, busca e recuperação de artefatos RAS.

Nas subseções que se seguem, estaremos interessados em apresentar a solução para os requisitos do RAS. Primeiramente faremos observações sobre os elementos envolvidos na criação do ambiente necessário para o suporte a RAS do ponto de vista de um repositório de reuso. Em seguida, apresentamos alguns conceitos importantes sobre a RAS para que possamos entender os elementos chaves envolvidos na identificação deste formato. Finalmente, apresentamos como se deu a implementação dos requisitos RAS e, adicionalmente, explicaremos alguns conceitos sobre o Archiva na medida em que eles forem necessários.

## Estendendo Soluções Existentes

Com a idéia do reuso em mente, nada mais esperado do que reutilizarmos algumas ferramentas que já executam parte do trabalho que nos é necessária em termos de infra-estrutura, mas que não contribuem diretamente para o suporte da RAS na implementação deste ensaio. Da escrita deste trabalho, dentre os gerenciadores de repositório Maven mais conhecidos está a ferramenta Archiva. Esta ferramenta provê importantes mecanismos de extensão (que, de certa forma, também a preparam para reuso) que a tornam muito atrativa para uso.

## Relação entre Geração e Consumo de Artefatos

Promover um ambiente de suporte a reuso implica em termos ferramentas para que a reutilização seja efetiva. Assim, não apenas necessitamos de ferramentas para gerar conteúdos passíveis de reutilização, como também ferramentas que, uma vez publicado, nos permitam encontrar esse conteúdo. Felizmente, o problema da pesquisa dos elementos já é endereçado pela ferramenta Archiva, bastando que nos preocupemos em colocar essas informações eficientemente dentro da ferramenta. O diagrama da Figura 4.1 demonstra a relação entre os atores da reutilização e as ferramentas de interesse do nosso domínio.



Figura .: Relação dos elementos do domínio

Dentre os relacionamentos identificados na Figura 4.1 destacamos primeiramente o relacionamento entre o desenvolvedor e as ferramentas que geralmente o circundam num ambiente de desenvolvimento, com a adição de elementos relativos ao domínio específico. Freqüentemente, os desenvolvedores optam por um ambiente integrado de desenvolvimento (*Integrated Development Environment*), ferramenta que, entre outros benefícios, promove o aumento da produtividade. É interessante que, para se ter uma experiência de reuso mais efetiva, a funcionalidade de geração de artefato RAS esteja integrada a esta ferramenta, de forma que o desenvolvedor não necessite aplicações adicionais para realizar o processo de reutilização de *software*, fazendo com que seja um processo natural.

Assim, o GeradorRAS é uma ferramenta destinada a auxiliar a geração do conteúdo RAS em um ambiente integrado, podendo ser visto como um facilitador na adoção do reuso. Entretanto, esta ferramenta não é de todo essencial, pois um desenvolvedor conseguiria imitar sua funcionalidade utilizando-se de ferramentas complementares, como um editor XML e um compactador de arquivos. Por este motivo, vemos esta ferramenta como um trabalho futuro.

Sendo assim, concentramos nosso foco em um elemento de extrema importância para resolver o problema do suporte ao reuso: o ConsumidorRAS. Este é responsável por tornar o artefato RAS um elemento passível de pesquisa e, portanto, acessível a outros usuários. Este será o principal sistema envolvido no compartilhamento do conhecimento pelo desenvolvedor, já que ele é o responsável pela integração do formato RAS com a base de dados já existente no repositório, permitindo que demais usuários daquele repositório o encontrem.

## O artefato RAS

O artefato RAS é um arquivo que contém elementos passíveis de reuso, sejam eles documentos, código, bibliotecas ou arquivos no formato binário. A especificação RAS estabelece dois tipos de empacotamento. Estamos interessados no caso em que ele é compactado utilizando Zip como um arquivo contendo todos os elementos necessários para sua interpretação.

[http://www.pkware.com/documents/casestudies/APPNOTE.TXT][RFC1952]

Cada arquivo RAS contém zero ou mais arquivos *XML Schema*, um arquivo manifesto na raiz, um ou mais arquivos-artefatos. O que diferencia este arquivo de outro pacote qualquer é a existência de um descritor que referencia cada um dos elementos do pacote. Este descritor é chamado *rasset.xml*. A Figura 4.2 mostra um exemplo de um arquivo RAS com seu descritor.



Figura .: Arquivo RAS como um arquivo ZIP.

Neste documento os termos **arquivo de manifesto**, **arquivo descritor** e **rasset.xml** serão usados indistintamente.

O arquivo de manifesto segue um determinado esquema que define o tipo dos artefatos que estão sendo descritos. Este esquema é denominado perfil (*profile*). O RAS define três perfis básicos derivados da própria especificação. Neste documento estaremos preocupados apenas com o mais genérico deles, chamado de Perfil Padrão (*Default Profile*).

O perfil padrão define um elemento principal chamado *asset*, que descreve o artefato RAS. O elemento *asset* contém outros elementos, em ordem, profile, description, classification, usage, solution e um ou mais elementos *related-asset*. Destes elementos, apenas *description* e *classification* serão utilizados neste trabalho. O primeiro trata de uma descrição mais textual e completa sobre oativo em questão. Já o segundo estabelece elementos para classificação do *asset*, como descritores e informações sobre o contexto onde o *asset* está inserido.

O elemento *context* do elemento *classification* pode nos dar informações importantes sobre o ativo em questão. A estrutura deste elemento é mostrada na Figura 4.3.



Figura .: elemento *context* de /asset/*classification*.

A função deste elemento é localizar o ativo junto a um ou mais contextos de uso. Um contexto pode conter informações sobre sob quais circunstâncias o ativo foi criado, qual a tecnologia utilizada, etc. Este elemento possui dois outros elementos que podem ser úteis no caso de pesquisa. São eles o *description* (logo abaixo do elemento *context*) e o *descriptor.* Estes são os elementos nos quais estaremos interessados em indexar, já que fornecem informações muito relevantes relativas ao ativo.

Alguns atributos do elemento raiz *asset* são especialmente importantes pra nós. Estes elementos são: *name*, *id*, *version* e *short-description*. A função destes elementos no documento se dá como segue:

* *name*: identifica o ativo em poucas palavras de interesse para consumo humano.
* *id*: contém um identificador único global a ser usado por ferramentas para distinguir ativos entre si.
* *version*: usado para comparar dois ativos com o mesmo *id*.
* *short-description*: uma descrição de aproximadamente uma linha sobre o ativo em questão, para visualização humana.

Todos estes elementos possuem informações relevantes para o caso de pesquisa de ativos e, portanto, devem ter seu conteúdo indexado.

## Suporte do formato RAS no Archiva

O passo principal no caminho à compatibilidade com a RAS é o suporte do tipo de arquivo nela especificado em um repositório de reuso. Para tanto, optamos por usar um gerenciador de repositório já existente, chamado Archiva, pois a ferramenta permite certo grau de extensibilidade. Ainda que não presente no RAS, modificações visuais também serão necessárias nesta ferramenta de modo que as informações do arquivo *.ras* sejam introduzidas e apresentadas de maneira desejável através da interface com o usuário. Além disso, a especificação RAS sugere que um repositório compatível deve suportar URLs particulares de pesquisa.

Nos próximos tópicos descreveremos alguns conceitos importantes para compreensão das decisões de projeto feitas e como se deu o design da solução de suporte ao formato RAS na ferramenta Archiva. Primeiramente, apresentamos a forma como o Archiva permite que desenvolvedores promovam suporte a tipos de arquivos diferentes. Em seguida, a fonte de informação da base de dados do Archiva é descrita brevemente. Dando continuidade, mostramos algumas características do Archiva que nos ajudam a resolver o problema proposto e, finalmente, apresentamos a nossa solução ao problema proposto.

### Consumidores Archiva

O gerenciador Archiva possui uma estrutura preparada para extensão através de uma API para desenvolvimento de *plugins* de suporte a tipos desconhecidos de arquivos. Estes *plugins* são chamados Consumidores (*Consumers*). Consumidores são components que consomem ou processam um artefato. Os consumidores podem ser de dois tipos: Consumidores de Conteúdo de Repositório (*Repository Content Consumers*) e Consumidores de Base de Dados (*Database Consumers*).

Os primeiros são responsáveis por processar artefatos durante o *scanning* do repositório. Neste caso, para cada artefato encontrado no repositório, cada consumidor disponível o processa.

Analogamente, o segundo tipo consome ou processa artefatos durante o *scanning* da base de dados. É divido também em dois grupos. O primeiro, chamado de consumidores de artefatos não-processados (*Unprocessed Artifact Consumers*), é responsável por aqueles artefatos já indexados que não foram processados ainda, de modo que os detalhes sobre aquele artefato não foram processados e salvos na base de dados. O segundo grupo é responsável por limpar a base de dados (de referências mortas, por exemplo), e são chamados de consumidores limpadores de artefatos (*Artifact Cleanup Consumers*). [http://archiva.apache.org/docs/1.1.3/adminguide/consumers.html]

Os consumidores serão de suma importância para nós pois será através deles que conseguiremos suportar um novo formato de arquivo dentro do repositório.

### O Arquivo POM

O arquivo POM – *\*.pom* – é um elemento importante na representação dos dados do repositório Maven pelo Archiva. Apesar disso, neste tópico não apresentaremos a estrutura do arquivo POM por não ser do objetivo deste trabalho. Estaremos interessados apenas nas informações relativas à formação do identificador universal do artefato. Originalmente, este arquivo é utilizado pelo repositório Maven como uma representação em XML de informações sobre um projeto Maven. Seu nome é uma sigla para Modelo de Objeto de Projeto (*Project Object Model*). Quando nos referimos à informação de modelo de projeto, queremos fazer referência à informação sobre o artefato em um nível mais alto, além da mera coleção de arquivos contendo código-fonte.

Um projeto contém arquivos de configuração, assim como desenvolvedores envolvidos e suas atribuições, um sistema de rastreamento de defeitos, uma organização e licenças, a URL onde o projeto reside, as dependências do projeto e todas as outras pequenas peças que vêem juntas para “dar vida” ao código. Neste contexto, o arquivo POM é um lugar para colocar informações sobre todas as coisas relativas ao projeto. Na verdade, em um mundo Maven, um projeto pode não conter nenhum código, apenas meramente um POM.

No POM, *groupId, artifactId* e *version* são os únicos campos requeridos. Os três campos funcionam como um endereço e marcação temporal em apenas um. Eles marcam um lugar específico no repositório, atuando como um coordenador de sistemas para projetos Maven. Abaixo descrevemos os campos que podem ser utilizados para identificar um artefato unicamnete:

* **groupId**: Este é geralmente único entre as organizações para um projeto. Por exemplo, todos os principais artefatos Maven possuem um mesmo *groupId* org.apache.maven. O seu valor não necessariamente usa a notação com pontos, como por exemplo, o projeto **junit**.
* **artifactId**: Este campo é geralmente o nome pelo qual o projeto é conhecido. Ainda que o *groupId* seja importante, pessoas dentro do grupo raramente vão mencioná-lo em discussões. Juntamente com o *artifactId*, criará uma chave que separa um projeto de outro qualquer no mundo.
* **version**: Esta é a última parte do quebra-cabeça da nomenclatura. *groupId:artifactId* denota um projeto, mas não consegue delinear qual encarnação daquele projeto estamos falando. Em resumo: o código muda, estas mudanças devem ser versionadas e este campo mantém estas versões em linha.
* **packaging**: Agora que nós temos a estrutura do endereço, falta apenas um campo padrão para identificar completamente um artefato. Este é o tipo de artefato do projeto. Os valores atuais para *packaging* são *pom, jar, maven-pluginm ejb, ear, rar, par*. Estes definem uma lista de objetivos executados para cada estágio do ciclo de vida de construção correspondente para uma estrutura de pacote.
* **classifier**: Ocasionalmente pode-se encontrar um quinto elemento na coordenada de endereço, o qual denominamos *classifier*. É suficiente saber que estes tipos de projetos são mostrados como *groupId:artifactId:packaging:classifier:version*.

REF: [http://maven.apache.org/pom.html#What\_is\_the\_POM]

Além desses campos, estaremos trabalhando também com os elementos *name* e *description*. Ainda, nos tópicos que se seguem, daremos um novo significado para o campo *packaging*, já que a este não possui um valor para artefatos RAS. Este campo será particularmente importante para nós, uma vez que este campo é essencial para identificação dos artefatos que possuem modelo de projeto. Como veremos também, todas estas informações estão representadas por uma classe dentro do Archiva chamada ArchivaProjectModel. Estas informações estão presentes dentro da base de dados do Archiva e são utilizadas em sua visualização como na Figura 4.7.

Podemos perceber uma relação da lista de campos acima com os campos presentes no arquivo descritor do RAS. Para os campos *artifactId*, *version*, *name* e *description* existe uma relação direta com *id*, *version*, *name* e *short-description* do *rasset.xml*, respectivamente. De fato, há muita semelhança entre estes dois descritores. Entretanto, o POM está intimamente ligado ao sistema de construção do ativo em questão, indicando uma forte relação com o código fonte. Já o descritor RAS não possui esta conotação, uma vez que o formato RAS não é destinado somente a código fonte, mas a qualquer tipo de ativo dentro do domínio de reuso.

Já que o modelo de projeto (POM) e o Perfil Padrão do RAS não possuem os mesmos campos, a Tabela 4.1 propõe o mapeamento para alguns campos cuja relação é trivial.

Tabela 4.1: Mapeamento POM vs. RAS

|  |  |
| --- | --- |
| **Elemento POM** | **Elemento RAS** |
| /project/groupId | /asset/@id |
| /project/artifactId | /asset/@id |
| /project/version | /asset/@version |
| /project/description | /asset/@short-description |
| /project/name | /asset/@name |

### Colocando artefatos RAS no Archiva

Felizmente o envio de artefatos RAS em si para o Archiva não é uma tarefa que necessite alteração. O Archiva já fornece uma interface para que o usuário submeta artefatos ao repositório. Entretanto, os campos desta interface estão intimamente ligados com a representação do artefato no repositório já que podemos observar que os campos são exatamente aqueles descritos como obrigatórios pelo arquivo POM. Neste tópico, descrevemos esta interface, bem como o significado que demos aos seus campos.

A interface de envio de artefato é apresentada na Figura 4.4. Atualmente, esta interface se apresenta com informações obrigatórias sobre o artefato, como *group id, artifact id, version* e *packaging*, informações estas indispensáveis para formar o endereço do artefato dentro do repositório, como colocado na subseção anterior. Sendo o *Archiva* um gerenciador para repositórios *Maven*, a identificação dos elementos segue aquela descrita pelos elementos do *POM*.

No nosso caso, algumas dessas informações estão presentes no arquivo descritor do artefato RAS ou poderiam ser inferidas. Infelizmente, o Archiva não fornece uma maneira segura de pré-processarmos o conteúdo do arquivo antes de ele ser copiado para sua localização final, de modo que nos cabe apenas assegurar que as informações do índice e da base de dados estejam corretas.



Figura .: Tela de envio de artefato do Archiva

#### Envio do Arquivo

Uma vez que o usuário do repositório dispara o envio do arquivo, o Archiva transfere o conteúdo do arquivo para uma localização temporária e dá início ao processo de verificação de novo artefato. O diagrama UML da Figura 4.5 mostra a relação entre a ação de envio de artefato e os consumidores de repositório.

A ação *UploadAction* é invocada pelo método *doUpload* que, dentre outras coisas, executa uma ação sobre uma coleção de consumidores descobertos durante o processo de inicialização do Archiva através de um arquivo de configuração. Este processo se realiza na forma do padrão de projeto *Chain of Reponsability* [GOF]. No diagrama da Figura 4.6 incluímos duas entidades importantes no processo de reconhecimento de arquivos *.ras*: o *RasConsumer* e o *RassetReader*.

O método *doUpload* requer aquelas informações obrigatórias da tela mostrada na Figura 4.4. São elas *group id, artifact id, version, packaging* e *artifact file*. No nosso domínio, os campos *artifact id* e *version* estão representados na RAS, respectivamente, pelos elementos Id e Versão da Tabela 4.2. O campo *packaging* é fixo, com o valor “ras” para identificar que o arquivo tem este tipo. Resta apenas definir o valor de *group id*. Entretanto, a RAS não define um campo *group id*, de modo que adotaremos a convenção de repetir o valor de *artifact id* no campo *group id*. O campo *artifact file* especifica o arquivo a ser enviado ao repositório, no nosso caso, um arquivo .ras.

RAS é então copiado para a sua localização definitiva, mais especificamente para o caminho *repositório/Id/Id/Versão*. Em seguida, inicía-se o trabalho dos Consumidores, cuja chamada é desencadeada pelo próprio método doUpload.

Para que o artefato possa ser recuperado, é preciso criarmos uma referência para ele na base de dados do Archiva. Felizmente, o Archiva já possui um consumidor que analisa os arquivos e os coloca na base de dados e, portanto, não precisamos nos preocupar com essa parte. Entretanto, para que o arquivo fique disponível para pesquisa, é necessário atualizarmos o índice do Archiva para conter as informações referentes ao artefato. É importante ainda que o artefato tenha suas informações mostradas na tela do Archiva (Figura 4.7). Para isso é necessário que as informações possam ser interpretadas corretamente, de modo que precisamos inserir na base de dados um modelo descritivo do artefato. Este modelo é chamado modelo de projeto.



Figura .: Relação entre envio de artefato e consumidores de artefato incluindo RasConsumer

#### Indexação do Arquivo

A indexação do arquivo deve ocorrer de acordo com as necessidades de pesquisa do mesmo. Esta tarefa é feita pelo RasConsumer, um consumidor da classe Consumidores de Conteúdo de Repositório. A subseção 9.1 da RAS define uma pesquisa por palavra chave e estabelece que:

Esta requisição [procura por palavra chave] deve procurar pelo menos os metadados do artefato. Em particular, nome, id, versão, descrição curta, descrição e seção classificação. [RAS, p.99]

Os elementos citados podem ser encontrados no Perfil Padrão e são definidos pelas expressões XPath na Tabela 4.2:

Tabela .: Relação de elementos do Perfil Padrão a serem indexados

|  |  |
| --- | --- |
| **Elemento** | **XPath equivalente** |
| Nome do Ativo | /asset/@name |
| Identificador do Ativo | /asset/@id |
| Versão do Ativo | /asset/@version |
| Descrição breve do Ativo | /asset/short-description |
| Descrição do Ativo | /asset/description//text() |
| Descrição dos Contextos | /asset/classification//context/description/text() |
| Descritores dos Contextos | /asset/classification//context//descriptor/text() |

Na tabela acima está assumido que o XML em questão usa o *namespace* padrão para identificar o RAS. Entretanto, isto deverá ser levado quando da leitura do mesmo, que será ajustada de acordo com o prefixo utilizado.

Uma vez desencadeado o trabalho dos consumidores, o arquivo que foi enviado para a base e agora se encontra em sua localização definitiva será processado por cada consumidor de repositório disponível. O responsável por indexá-lo corretamente será o nosso *RasConsumer*. Mas para extrairmos o conteúdo passível de indexação, teremos de descompactar o rasset.xml de dentro do artefato para executar as pesquisas relacionadas na Tabela 4.2. Para isso, descompactamos o rasset.xml para uma localização temporária junto ao diretório do artefato. Após, as pesquisas são aplicadas e os resultados concatenados e salvos em memória. O arquivo temporário é excluído, e os resultados retornados ao *RasConsumer*, que os encaminha para o indexador do Archiva. O processo de extração dos valores é de responsabilidade do *RassetReader*.

A inserção na base de dados já é feita por um consumidor genérico, chamado *ArtifactUpdateDatabaseConsumer*. Ele processa os *checksums* que permitem que o arquivo seja localizado pela interface *Find Artifact* do Archiva. Além disso, seta outras informações relevantes do artefato e então o salva na base de dados.

#### Inserção do Modelo de Projeto do Artefato na Base de Dados

Apenas as informações de índice e a presença do artefato na base de dados não garantem que o artefato possa ser visualizado como na Figura 4.7. O Archiva diferencia a informação do artefato em si daquela que é visualizada pela interface, tratando-as de maneira separada. A representação das informações visuais do artefato é chamada modelo de projeto e também fica presente na base de dados. Entretanto, sua inserção ocorre num outro momento, já que o consumidor que a tratará não é da mesma classe daquela do RasConsumer.

As informações do modelo de projeto são inseridas por consumidores de base de dados. No Archiva, as informações visuais do artefato são guardadas pelo pom.xml. Este arquivo é, assim como o rasset.xml, um descritor para o artefato. Este descritor, definido especialmente para a ferramenta Maven, contém elementos semelhantes ao rasset.xml. Atualmente, no Archiva, há um consumidor que verifica as informações do arquivo pom.xml e coloca na base o seu respectivo modelo de projeto. Entretanto, não existe consumidor similar que realize este processo para os descritores RAS. Portanto, uma vez que o artefato esteja indexado e presente na base, é preciso que o modelo deste artefato RAS esteja descrito da base de dados para que sua visualização seja correta.



Figura .: Representação UML para o RasDatabaseConsumer

Para inserirmos as informações do artefato corretamente na base de dados, é necessário construirmos um consumidor de artefatos não-processados, seguindo o mesmo princípio realizado pelo consumidor que processa o pom.xml (*ProjectModelToDatabaseConsumer*). A execução deste tipo de consumidor é disparada pela ação de *scanning* de base de dados. Na Figura 4.6, o consumidor que processará as informações do modelo RAS está representado pelo elemento RasDatabaseConsumer.

A ação de scanning não é executada imediatamente. Quando requisitada, a mesma entra em uma fila de execução, e aguarda um disparador temporal. No Figura 4.6, a classe que executa efetivamente a atualização da base de dados é a *ArchivaDatabaseUpdateTaskExecutor*. Ela chama o método *updateAllUnprocessed* da classe *JdoDatabaseUpdater*, que novamente, se utilizando do *Chain of Responsability*, chama os respectivos métodos de execução dos consumidores para cada arquivo novo da base.

Embora seja um consumidor diferente do RasConsumer, o RasDatabaseConsumer utiliza a mesma classe para extração temporária do rasset.xml do artefato RAS.

O RasDatabaseConsumer busca as informações relevantes do rasset.xml e, utilizando o mapeamento definido na **Error! Reference source not found.**, cria um *ArchivaProjectModel* para acomodar as informações do artefato RAS. Em seguida, ele se utiliza da classe *JdoProjectModel*  para salvar este modelo de projeto na base de dados.

### Adaptação para Apresentação dos Resultados

A apresentação dos resultados na tela do Archiva não é necessariamente um requisito para atender à especificação RAS. Entretanto, num ambiente de suporte a reuso, para que o reuso aconteça é necessário que os desenvolvedores tenham uma forma de pesquisar e verificar informações sobre esses artefatos presentes no repositório. Ainda, por se tratar de uma ferramenta já presente no mercado, é importante manter a consistência das visualizações nela presentes. Portanto, foi necessário que tornássemos a interface com o usuário do Archiva ciente do formato de arquivo especificado pela RAS.

Na Figura 4.7 podemos visualizar o artefato JUnit. São mostradas na tela principal as informações do artefato, como identificador e versão, bem como uma breve descrição. Na caixa flutuante à direita temos os links para recuperar o artefato e o seu arquivo descritor.

A interface com o usuário do Archiva consegue fornecer essas informações porque possui uma forma de ler o descritor deste arquivo presente no repositório. Este descritor é o pom.xml, como abordado anteriormente. Assim, a Figura 4.7 apresenta, na verdade, uma representação deste descritor.

Para que o arquivo RAS seja visualizado corretamente, é necessário que o Archiva procure por pelo seu descritor. Como o construímos na subseção anterior, agora temos de fazer o Archiva encontrá-lo corretamente.



Figura .: Tela de visualização de artefato JUnit do Archiva

A busca do descritor está associada à busca do artefato em si. Esta é iniciada por uma ação na interface com o usuário do Archiva. A partir da opção *Browse*, é mostrado ao usuário uma opção de navegação. A navegação através do repositório inicia-se pela escolha do *group id* do artefato. Em seguida, o usuário escolhe o *artifact id* do artefato que deseja ver. Em última instância, o usuário é apresentado com as opções de versão daquele artefato. Um exemplo desde fluxo é apresentado pela Figura 4.8



Figura .: Navegação pelos artefatos do repositório

Quando o usuário seleciona uma versão, a classe *ShowArtifactAction* é ativada e busca as informações relativas ao modelo de projeto do artefato. A Figura 4.9 mostra um diagrama que representa a relação existente entre os participantes deste processo.



Figura .: Relação entre elementos de visualização do modelo de projeto.

Sendo o Archiva uma interface para repositório Maven, é razoável supor que ela está preparada para mostrar informações apenas daqueles artefatos que possuem o arquivo descritor do Maven (o pom.xml). A classe responsável por enviar as informações do modelo de projeto à interface com o usuário do Archiva é a ShowArtifactAction. Esta classe utiliza-se de um método da classe **DefaultRepositoryBrowsing** para procurar o artefato. O processo feito para achar tal modelo de projeto é, em primeiro lugar, encontrar o modelo de artefato *pom* associado com o artefato do qual se quer obter o modelo de projeto. Atualmente, se este artefato não é encontrado, a exibição do arquivo é abortada e o usuário vê uma página de erro.

Para que a visualização do artefato RAS seja completa, é necessário alterarmos **DefaultRepositoryBrowsing** para que busque as informações de modelo de projeto de artefatos RAS incondicionalmente, mesmo que não exista um arquivo *pom* associado. Afinal, essas informações foram colocadas na base de dados, mas sobre um *packaging* diferente – um *packaging* RAS. A mudança necessária é pesquisar os artefatos com o *packaging* “ras” caso não seja encontrada nenhuma instância com o *packaging* “pom”.

[]

### Recuperação e Pesquisa de Artefatos no Archiva

O Archiva possui meios para pesquisa e recuperação de artefatos. Entretanto, estas funções precisam estar disponíveis conforme na seção 9 da especificação RAS que estabelece URLs para cada uma dessas operações. Para a primeira delas, é necessário primeiramente compreender como o sistema de pesquisa de artefatos funciona. Para a segunda operação, o Archiva já fornece uma interface HTTP para acesso, sendo que tudo o que precisamos fazer é construir um atalho para que a partir da URL especificada no RAS, possamos ativar esta já disponível.

Ambas as operações terão de ser implementadas no Archiva através de Java Servlets. Esta tecnologia provê desenvolvedores com um mecanismo simples e consistente para estender a funcionalidade de um servidor Web e para acessar sistemas de negócios existentes. [http://java.sun.com/products/servlet/]

#### Pesquisa de Artefatos

A pesquisa de artefatos no Archiva está implementada através de um formulário na interface com o usuário. A partir do menu inicial, o usuário pode escolher a opção “Search” e informar, no campo que aparece na tela, valores de pesquisa. A confirmação da ação aciona a classe SearchAction do Archiva, que faz a pesquisa do artefato e retorna as informações sobre ele.



Figura .: Diagrama Simplificado para operação Search

O diagrama da Figura 4.10 mostra a relação entre as classes envolvidas no processo de pesquisa de artefatos. Neste diagrama, identificamos três elementos importantes para a busca das informações de pesquisa. O primeiro deles é o método que dá início ao processo de pesquisa – **quickSearch** da classe **SearchAction**. Ele utiliza como parâmetro principal o atributo da classe chamado **q** que, por sua vez, contém as palavras de pesquisa.

A lógica de pesquisa está presente em uma classe chamada **DefaultCrossRepositorySearch**. Esta classe usa o indexador padrão do Archiva através do método **searchTerm** e instancia um objeto da classe MultiSearcher que executa a pesquisa em última instância. Os resultados são coletados e o retorno desta chamada, feita pela classe **SearchAction**, é colocado no atributo **results**. Este, por usa vez, é utilizado pela interface com o usuário para mostrar as informações. Este elemento contém todos os registros encontrados na pesquisa, cada um deles representado pela classe **SearchResultHit**.

A especificação RAS não define uma formatação para o retorno dos resultados. Decidimos representar os dados de retorno com XML, já que este padrão nos permite especificar exatamente como o retorno será fornecido. Assim, a representação do termo Descritor de Ativo de Repositório (RepositoryAssetDescriptor) pode ser encontrada no apêndice A.

#### Procura por Palavra-Chave

Para a construção do nosso serviço de pesquisa, não seria apropriado utilizar diretamente a classe SearchAction por ela fazer parte de um *framework* de inversão de controle que automaticamente a relaciona com o código responsável pela interface com o usuário. Quando isso acontece, os atributos da classe são preenchidos pelo *framework* através das diretivas especificadas no código e em arquivos de configuração ou com informações obtidas em tempo de execução. Ao invés disso, utilizaremos diretamente a classe **DefaultCrossRepositorySearch**, da Figura 4.10, instanciando-a através dos métodos disponíveis pelo *framework*. Para suportar os métodos de pesquisa, utilizamos uma estrutura chamada Servlet, através da qual podemos processar uma requisição HTTP diretamente, instanciar os componentes necessários e realizar a pesquisa.

A classe **SearchResultHit** não fornece todas as informações de retorno de pesquisa necessárias para a conformidade com o padrão RAS. O resultado nos dá **groupId**, **artifactId**, **version** e URL, além o identificador do repositório (**repositoryId**). A especificação determina que a resposta do serviço deve conter o nome do artefato, uma breve descrição, a URL para download, o caminho lógico, a versão e um índice de certeza [RAS, pág. 105]. Portanto, estão faltando ainda o nome do artefato, sua descrição e o índice de certeza. Além disso, os resultados retornados pelo método **searchTerm** contém elementos indesejados, como aqueles artefatos que não seguem o formato RAS (não podemos esquecer que os repositórios Maven têm, em sua maioria, pacotes Java).

Podemos nos utilizar do modelo de projeto do artefato para buscar as informações de nome e descrição. Para tal, basta instanciarmos um objeto da classe JdoArchivaDAO, vista anteriormente, para que através dela consigamos acesso ao modelo de projeto. Através desta *Abstract Factory* temos acesso a um objeto da classe JdoProjectModelDAO e, através do método *getProjectModel*, acessamos o modelo de projeto do artefato, além de já filtrarmos os artefatos que não nos interessam. Este mecanismo é disparado através de uma requisição HTTP que é processada pelo *servlet* RasSearchServlet no método *doGet*. O diagrama da Figura 4.11 mostra um esquemático para a seqüência acima descrita.



Figura .: Diagrama de Seqüência da operação de pesquisa

#### Procura por Caminho Lógico

/SearchByLogicalPath?path=<caminho-lógico>

# Conclusão

A solução descrita na subseção 4.1 resolveu o problema da falta de ferramentas para geração de arquivos no formato RAS.

A solução descrita na subseção 4.2 permitiu que o formato RAS fosse reconhecido dentro de uma ferramenta de suporte a reuso.

A necessidade das métricas.

A aceitação do RAS no mercado.

glossário

Esse item é opcional. Se houver glossário, apresentar depois das referências.

anexo A <Descrição do anexo>

Destinam-se à inclusão de informações complementares ao trabalho, mas que não são essenciais à sua compreensão. Os Apêndices devem apresentar material desenvolvido pelo próprio autor, formatado de acordo com as normas. Já os Anexos destinam-se à inclusão de material como cópias de artigos, manuais, etc., que não necessariamente precisam estar em conformidade com o modelo, e que não foram desenvolvidos pelo autor do trabalho. A contagem das páginas nos Apêndices e Anexos segue normalmente. Nos Anexos os números não precisam ser indicados, a não ser na página inicial de cada um.

**No caso de haver apenas um anexo, não utiliza-se as letras para enumerá-los. Usa-se a palavra ANEXO no singular.**

anexo b <EXEMPLO dE anexo>

Xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

apêndice Esquema XSD Para Descritor de ativo de Repositório

<?xml version="1.0" encoding="utf-8"?>

<xs:schema id="RAS\_RepositoryAssetDescriptor"

targetNamespace="http://www.ufrgs.inf.br/RAS/RepositoryAssetDescriptor"

elementFormDefault="qualified"

xmlns="http://www.ufrgs.inf.br/RAS/RepositoryAssetDescriptor"

xmlns:mstns="http://www.ufrgs.inf.br/RAS/RepositoryAssetDescriptor"

xmlns:xs="http://www.w3.org/2001/XMLSchema"

xml:lang="pt-br"

>

<xs:element name="RepositoryAssetDescriptor">

<xs:complexType>

<xs:sequence>

<xs:element name="Name" type="xs:string">

<xs:annotation>

<xs:documentation>Mapeia para o atributo name do elemento asset do RAS</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name="Description" type="xs:string">

<xs:annotation>

<xs:documentation>Mapeia para o atributo short-description do elemento asset do RAS</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name="Url" type="xs:anyURI">

<xs:annotation>

<xs:documentation>URL para o ativo (recuperando esta URL deve trazer o arquivo .ras)</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name="LogicalPath" type="xs:anyURI">

<xs:annotation>

<xs:documentation>Caminho lógico do ativo no repositório</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name="Version" type="xs:string">

<xs:annotation>

<xs:documentation>Mapeia para o atributo version do element asset do RAS</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name="Ranking" type="xs:int">

<xs:annotation>

<xs:documentation>Valor entre 0 e 100, com 100 sendo o melhor acerto</xs:documentation>

</xs:annotation>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="RepositoryAssetDescriptorCollection">

<xs:annotation>

<xs:documentation>Coleção de descritores de repositório</xs:documentation>

</xs:annotation>

<xs:complexType>

<xs:sequence>

<xs:element ref="mstns:RepositoryAssetDescriptor" minOccurs="1" maxOccurs="unbounded"/>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:schema>