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

2.3 Desenvolvendo com Reuso e para Reuso 15

3 Repositórios de Software 16

3.1 Repositório de Gerência de Configuração 16

3.1.1 Características Chaves **Erro! Indicador não definido.**

3.1.2 Aplicação Prática **Erro! Indicador não definido.**

3.2 Repositório de Reuso 16

3.2.1 Características Chaves **Erro! Indicador não definido.**

3.2.2 Aplicação Prática 16

3.3 Ciclo de Vida dos Artefatos de Software 17

3.4 Ferramentas Existentes Erro! Indicador não definido.

3.4.1 Gerência de Configuração **Erro! Indicador não definido.**

3.4.2 Reuso **Erro! Indicador não definido.**

3.5 Ferramentas que Suportam o RAS Erro! Indicador não definido.

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!!!! **Erro! Indicador não definido.**

4 Promovendo o Reuso de Software utilizando RAS 19

4.1 Estendendo Soluções Existentes 20

4.2 O artefato RAS 22

4.3 Suporte do formato RAS no Archiva 24

4.3.1 Consumidores Archiva 24

4.3.2 O Arquivo POM 24

4.3.3 Colocando artefatos RAS no Archiva 26

4.3.4 Adaptação para Apresentação dos Resultados 31

4.3.5 Recuperação e Pesquisa de Artefatos no Archiva 33

5 Conclusão 38

glossário 39

anexo A <Descrição do anexo> 40

anexo b <EXEMPLO dE anexo> 41

apêndice Esquema XSD Para Descritor de ativo de Repositório 43

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 3.1: Repositório de Gerência de Configuração versus Repositório de Reutilização 16

Figura 3.2: Integração dos Repositórios 17

Figura 4.1: Infra-estrutura de suporte ao padrão RAS 19

Figura 4.2: Relação dos elementos do domínio 21

Figura 4.3: Arquivo RAS como um arquivo ZIP. 22

Figura 4.4: elemento *context* de /asset/*classification*. 23

Figura 4.5: Tela de envio de artefato do Archiva 27

Figura 4.6: Relação entre envio de artefato e consumidores de artefato incluindo RasConsumer 28

Figura 4.7: Representação UML para o RasDatabaseConsumer 30

Figura 4.8: Tela de visualização de artefato JUnit do Archiva 31

Figura 4.9: Navegação pelos artefatos do repositório 32

Figura 4.10: Relação entre elementos de visualização do modelo de projeto. 33

Figura 4.11: Diagrama Simplificado para operação Search 34

Figura 4.12: Diagrama de Seqüência da operação de pesquisa por palavra-chave 36

LISTA DE TABELAS

[Tabela 4.1: Mapeamento POM vs. RAS 26](#_Toc232187134)

[Tabela 4.2: Relação de elementos do Perfil Padrão a serem indexados 29](#_Toc232187135)

[Tabela 4.3: Requisições que fazem sentido para o contexto do Archiva 36](#_Toc232187136)

RESUMO

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

Promoting Software Reuse Through RAS

ABSTRACT

**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 necessárias. Tais custos aumentados não 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.

## Desenvolvendo com Reuso e para Reuso

O reuso no processo de desenvolvimento pode ser abordado em duas diferentes práticas. Portanto, é interessante salientar a diferença entre desenvolvimento de software com reuso e desenvolvimento de software para reuso.

O desenvolvimento com reuso é a abordagem que tenta maximizar o reuso de software existente. Uma vantagem óbvia desta abordagem é que o custo total de desenvolvimento deve reduzir. Menor quantidade de elementos de software deve ser especificada, projetada, programada e validada. Entretanto, redução de custo é apenas uma das vantagens potenciais.

A adoção do reuso em tal caso pode ser feita num processo de desenvolvimento logo após a fase de especificação. Após aquela fase, os especialistas procurariam por componentes reusáveis que completassem a especificação e os incorporariam no desenvolvimento. Entretanto, para que isto seja efetivo é necessário satisfazer algumas condições: o custo de procura de componentes seja relativamente baixo; os componentes devem se comportar de forma especificada e devem ser confiáveis; além disso, deve existir documentação associada aos componentes para ajudar o desenvolvedor que executa o reuso a entender o que foi construído e, em último caso, ser capaz de adaptar a uma nova aplicação.

O reuso sistemático requer um catálogo apropriado e uma base documentada de componentes reusáveis. Uma concepção errônea é a de que estes componentes estão disponíveis em sistemas existentes. Na verdade, componentes criados como parte de sistemas existentes dificilmente reutilizados de forma trivial. Estes componentes estão ligados aos requisitos do sistema do qual eles fazem parte originalmente. Para serem reusados, eles necessitam ser generalizados para satisfazer um conjunto maior de requisitos. Assim, o desenvolvimento para o reuso envolve a adaptação dos componentes existentes para torná-los reusáveis.

Adaptar um componente para que ele seja reusável pode envolver diferentes tipos de mudanças: padronização de nomenclatura, adição ou remoção de operações que são específicas de um domínio de aplicação ou ainda a identificação de exceções que este componente pode vir a gerar. Além da adaptação e geração do componente, é necessário verificar a qualidade do mesmo. Todos estes processos se realizam após o componente original ter sido pronto, adicionando passos ao processo de desenvolvimento padrão.

## Benefícios do Reuso de Software

Reuso sistemático no processo de desenvolvimento de software oferece também outras vantagens:

* **Confiabilidade do sistema aumentada.** Componentes reusados, que foram exercitados em sistemas já funcionais, devem ser mais confiáveis que novos componentes. Estes componentes foram testados em sistemas operacionais e foram, portanto, expostos a condições reais de operação.
* **Risco do processo reduzido.** Se um componente existe, há menos incerteza no custo de reutilização daquele componente do que no custo de desenvolvimento de um novo. Este é um fator importante para os gerentes de projeto já que isso reduz incertezas nas estimativas de custo de projeto. Isto é particularmente verdade quando componentes relativamente grandes como subsistemas são reusados.
* **Uso efetivo pode ser feito de especialistas.** Ao invés de especialistas de aplicações fazerem o mesmo trabalho em projetos diferentes, estes especialistas podem desenvolver componentes reusáveis que encapsulam seus conhecimentos.
* **Padrões organizacionais podem ser expressos em componentes reusáveis.** Alguns padrões, como os padrões de interface com usuário, podem ser programados como um conjunto padrão de componentes. Por exemplo, componentes reusáveis podem ser desenvolvidos para construir menus em uma interface com o usuário. Todas as aplicações apresentam o mesmo formato de menu para os usuários. O uso de interfaces padrão com o usuário melhora a confiança já que os usuários são menos suscetíveis a erros quando apresentados com uma interface familiar.
* **O tempo de desenvolvimento de software pode ser reduzido.** Levar um sistema ao mercado tão cedo quanto possível é freqüentemente mais importante do que o custo de desenvolvimento. O reuso de componentes aumenta a velocidade de produção, pois ambos os tempos de desenvolvimento e validação devem ser reduzidos.

# Repositórios de Software

Segundo o dicionário Aurélio, repositório é um depósito, mas pode também ser visto como uma coleção. No tocante ao mundo do desenvolvimento de software, um repositório é um lugar seguro para guardar o software produzido. Freqüentemente eles são associados ao aumento da memória organizacional já que, ao servirem como pontos de referência, guardam todo o conteúdo produzido em uma organização. Como função básica, os repositórios permitem o armazenamento e recuperação de artefatos.

Os repositórios estiveram ligados por muito tempo ao processo de Gerência de Configuração. Neste sentido, eles são bem conhecidos, pois são abordados em diversos processos organizacionais como CMMI e MPSBR. Uma das partes chaves deste processo é chamada controle de versão, que define a necessidade de mantermos versões diferentes de um sistema de software ou controlar suas alterações. Neste sentido, entram em cena os sistemas de controle de versão, comumente chamados de repositórios de gerência de configuração.

Não menos importante, mas nem tão adotado sistematicamente, o processo de reuso também possui um repositório específico. Um repositório de reuso vem para auxiliar o processo de busca de componentes prontos que podem ser reusados. Como veremos, há um ponto na linha de desenvolvimento de software onde estes dois repositórios fazem parte do processo e se encontram.

Podemos nos perguntar a razão de termos dois repositórios separados quando ambos estarão guardando elementos de software (ou documentos importantes que se referem aos mesmos). A explicação é simples: o seu uso é diferente. A Figura 3.1 é uma representação das principais diferenças entre esses dois tipos de repositório.

Em um repositório de controle de versão para gerência de configuração, estaremos interessados no processo de desenvolvimento do software, ou seja, sua construção. Assim, estaremos lidando com os arquivos que compõem uma determinada solução ou que satisfazem um determinado caso de uso ou requisito. Estes estarão sendo freqüentemente alterados, dado que o processo de desenvolvimento acontece pela geração de várias versões de diversos arquivos, na maioria dos casos. Assim, os desenvolvedores estão cientes do que estão fazendo e onde as alterações estão localizadas, o que elimina a necessidade de pesquisa.

Analogamente, um repositório de reuso possui importância quando um artefato de software já se encontra em produção. Nesta ocasião, estaremos interessados em reusar **o componente**, programa, biblioteca, ou outra versão de artefato que esteja sendo produzida. Assim, a granularidade não é mais o arquivo que compõe o componente, mas o componente como um todo (ou um *build* do componente). Neste contexto, não estaremos interessados em alterar o componente, mas em reusá-lo, o que caracteriza um acesso de leitura. Entretanto, como saberemos se existe uma solução que serve exatamente as necessidades que temos no momento? Para tanto, é necessário pesquisar pelos artefatos disponíveis para saber qual deles completa os nossos requisitos.

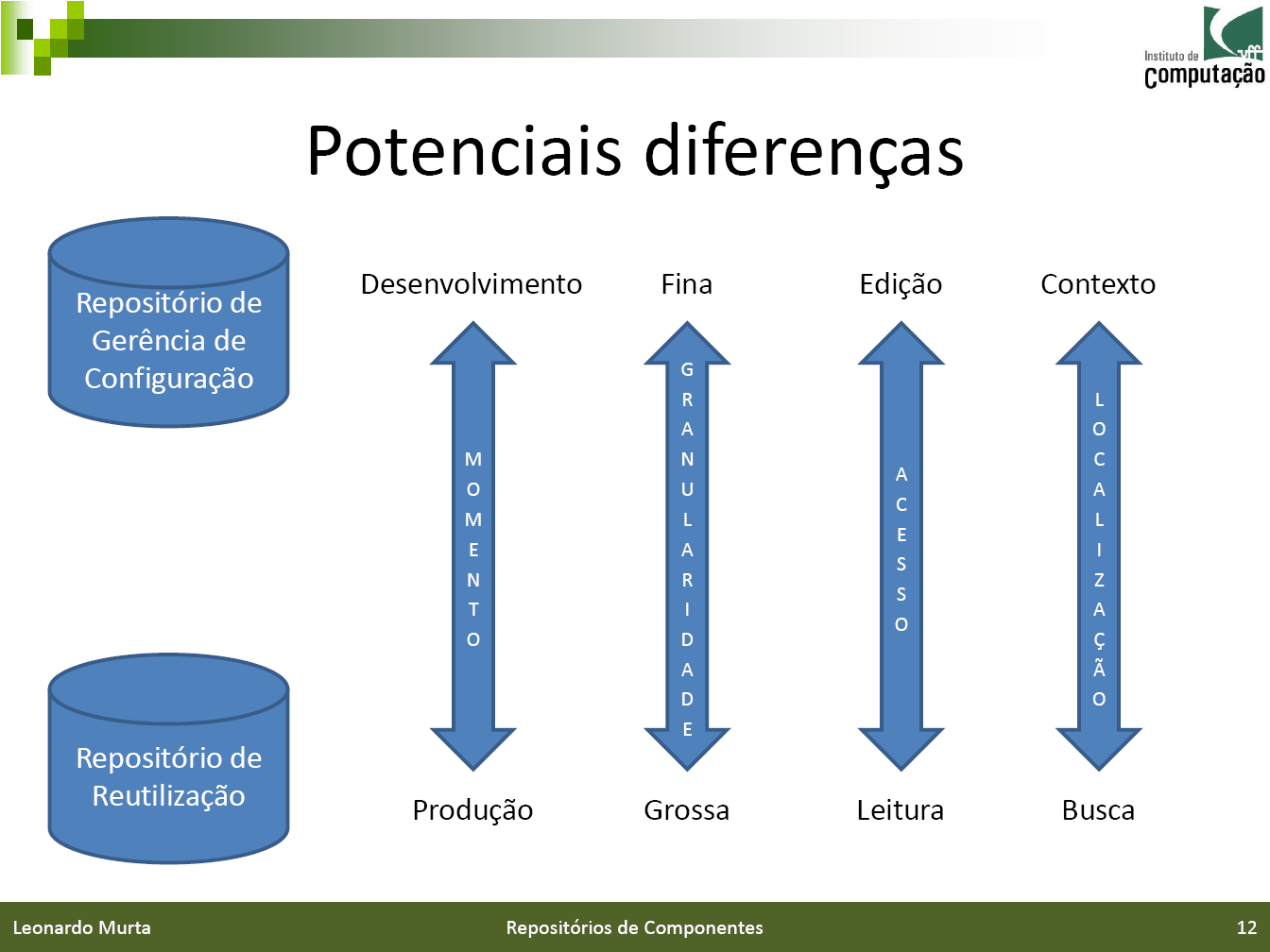


Figura .: Repositório de Gerência de Configuração versus Repositório de Reutilização

Agora que entendemos a diferença entre estes repositórios, podemos focar no problema deste trabalho tratando primeiramente dos repositórios de reuso. Nas subseções que seguem apresentaremos uma breve revisão sobre este conceito. Primeiramente, analisamos sua definição de acordo com a literatura disponível. Em seguida, evidenciamos a intersecção entre os processos de gerência de configuração e reutilização de forma a apresentarmos como se dá a ligação do processo de desenvolvimento com a reutilização. Adicionalmente, mostramos algumas ferramentas existentes no mercado.

## Repositório de Reuso

Um repositório de reuso é um lugar onde artefatos reusáveis são armazenados, juntamente com um catálogo de artefatos. Todos devem estar cientes que o repositório contém um importante *know-how* da organização, devendo ser capazes de acessar e usar o repositório de forma fácil .

Em poucas palavras, um repositório de reuso deve ser capaz de identificar artefatos, associar uma descrição a cada um deles e manter referências para os produtos que o compõe. Em mais detalhes, as funções de tal repositório são listadas abaixo:

* Identificação e descrição de artefato: cada artefato deve ser identificado em um formato homogêneo. É importante estabelecer e manter uma interface comum para os artefatos, para que os membros de uma organização possam se familiarizar com isso.
* Inserção de artefato: um usuário deve ser capaz de inserir um novo artefato no repositório.
* Navegação pelo catálogo: um usuário deve ser capaz de navegar o catálogo para acessar a descrição dos artefatos.
* Busca textual: um usuário deve ser capaz de procurar um texto contido em qualquer parte da descrição de um artefato no catálogo.
* Recuperação de artefato: dado um artefato definido, um usuário deve ser capaz de recuperar uma cópia do artefato.
* Organização e busca: navegação pelo catálogo é claramente insuficiente para identificar um artefato quando o repositório tem um número grande de artefatos, enquanto que a pesquisa textual pode consumir muito tempo. Diversas técnicas podem ser usadas nesses casos para organizar a descrição dos artefatos em catálogos e facilitar a pesquisa.
* Histórico: cada descrição de artefato deve gravar o autor, data de criação e de atualização, e uma lista de modificações. O uso do artefato também deve ser rastreado.
* Métricas: o repositório pode automatizar a coleta de dados para derivar métricas.
* Controle de acesso: cada uma das funções acima deve ser disponível apenas para perfis selecionados. Por exemplo, o direito de inserção deve ser garantido apenas aos gerentes de reuso, enquanto que navegação e recuperação podem ser garantidas a todos os usuários.
* Gerenciamento de versão: múltiplas versões do mesmo artefato devem ser definidas e gerenciadas, e seus relacionamentos gravados.
* Controle de alterações: procedimentos para requisitar, discutir, aceitar e programar alterações para os artefatos devem ser definidas e garantidas por um suporte automático.
* Notificação de alterações: alterações em artefatos e no repositório de reuso devem ser notificadas, com suporte automático, para todos os potenciais usuários.

Nem todas estas funções precisam necessariamente estar presentes em um repositório de reuso.

## O Papel do Repositório de Reuso no Processo de Gerência de Configuração

Por muito tempo o processo de reutilização de software foi feito *ad-hoc* ou de maneira não sistemática. Com o aumento da discussão sobre o tópico, foi visualizada uma conexão entre este processo e o próprio processo de desenvolvimento de software.

Embora reuso seja um item a parte, há uma conexão entre o seu ciclo de vida e o do processo de gerência de configuração.

## Ciclo de Vida dos Artefatos de Software

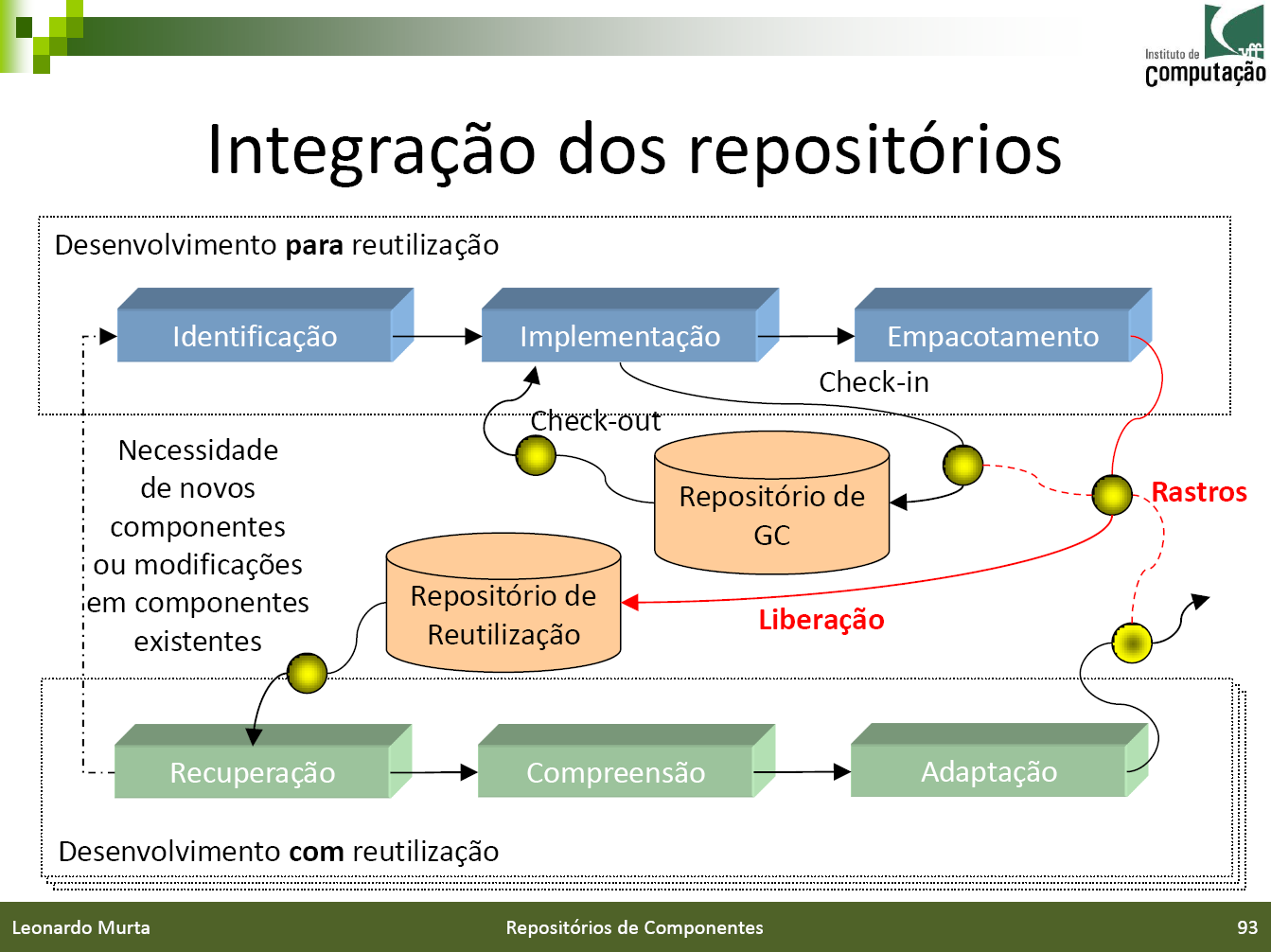


Figura .: Integração dos Repositórios

## Estado da ARte

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.

### Basic Asset Retrieval Tool e Component Repository

### Reusable Asset Manager

### ArcSeeker

## Archiva e Maven

Explicar quem são, talvez trazer alguns textos do Cap. 4 para cá - POM e consumers.

# Promovendo o Reuso de Software utilizando RAS

Podemos observar que um ambiente prático de suporte a reuso implica em termos ferramentas adequadas. Neste contexto, ferramentas como o gerenciador de projetos 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 promotora 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

Podemos intuir da descrição dos componentes da figura dois ambientes distintos. 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* dos artefatos que se encontram disponíveis. 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.

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 da perspectiva de um repositório de reuso. 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.

## Integrando 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. Precisamos de uma ferramenta que já possua um sistema de armazenamento, busca e recuperação de artefatos, para que nossa preocupação esteja restrita ao suporte da RAS. Estas três operações configuram um repositório de reuso.

Um repositório Maven pode ser um bom ponto de partida para nossa tarefa, dado que é uma ferramenta largamente utilizada atualmente. Entretanto, a ferramenta em si não disponibiliza uma interface web para gerenciamento dos seus recursos. Ainda assim, há ferramentas que o fazem. Da escrita deste trabalho, dentre os gerenciadores de repositório Maven mais conhecidos está a ferramenta Archiva. Esta ferramenta suporta as três operações necessárias e, além disso, provê importantes mecanismos de extensão (que, de certa forma, também a preparam para reuso) que a tornam muito atrativa para uso. A ferramenta funciona como um disseminador de informações sobre os artefatos disponíveis nos repositórios os quais estão sobre gerenciamento da mesma. Portanto, escolhemos como infra-estrutura principal o gerenciador de repositórios Archiva.

Para aprofundar a visão da Figura 4.1, detalhamos os envolvidos na tarefa de tornar o Archiva aderente à especificação RAS. Este detalhamento pode ser observado na Figura 4.2. Nesta figura destacamos primeiramente o relacionamento entre o desenvolvedor e as ferramentas que geralmente o circundam num ambiente de desenvolvimento integrado, com a adição de elementos relativos ao domínio específico do nosso problema. Freqüentemente, os desenvolvedores optam por um ambiente integrado de desenvolvimento (*Integrated Development Environment*), ferramenta que auxilia o desenvolvedor no processo de depuração dos seus programas e promove o aumento da produtividade. Para uma experiência de reuso mais efetiva, é interessante termos a funcionalidade de geração de artefato RAS, assim como a ferramenta de pesquisa de artefatos, integradas nesta ferramenta de forma que o desenvolvedor não necessite aplicações adicionais para realizar o processo de reutilização de *software*, fazendo com que este processo seja natural.



Figura .: Relação dos elementos do domínio

Nem todos os elementos dispostos nesta figura necessitam ser implementados. Na verdade, eles ilustram o que um ambiente RAS completo pode parecer. No diagrama, Archiva representa a ferramenta escolhida para a infra-estrutura deste trabalho. IDE representa o ambiente integrado de desenvolvimento de escolha do desenvolvedor. O Artefato RAS é um produto do Gerador RAS.

Dos elementos visionados, O Gerador RAS é uma ferramenta destinada a auxiliar a geração do conteúdo RAS em um ambiente integrado. Entretanto, esta ferramenta não é de todo essencial, pois um desenvolvedor conseguiria imitar sua funcionalidade utilizando-se de ferramentas complementares (ainda que com dificuldade), como um editor XML que execute validação do documento gerado e um compactador de arquivos. O Recuperador RAS, que é responsável por recuperar os artefatos, também pode ser imitado fazendo-se uso de um navegador para internet (o Archiva possui uma inteface HTTP). Por este motivo, estas ferramentas não serão o foco deste trabalho.

Mas, repositório de reuso que suporte a definição RAS é essencial para que consigamos um ambiente integrado aderente ao RAS. Sendo assim, concentramos nosso foco em um elemento de extrema importância para resolver o problema do suporte ao reuso com a especificação RAS: o Adaptador RAS. Este é responsável por tornar o artefato RAS compatível com as três operações citadas anteriormente suportadas pelo Archiva. Este será o principal (conjunto de) sistema(s) envolvido no compartilhamento do conhecimento pelo desenvolvedor, já que este sistema 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 o algoritmo 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 (rasset.xml) na raiz, um ou mais arquivos-artefatos. O que diferencia este arquivo de outro pacote qualquer é a existência deste descritor que referencia cada um dos elementos do pacote. A Figura 4.3 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: *profile*, *description*, *classification*, *solution*, *usage*, e um ou mais elementos *related-asset* (nesta ordem). Destes elementos, apenas *description* e *classification* serão utilizados neste trabalho.

Parte de nosso problema é ler estes dados do rasset.xml para que possamos indexar o artefato com base nessas informações. Assim, o primeiro elemento de interesse, *description*, trata de uma descrição mais textual e completa sobre oativo em questão. Já o elemento *classification* estabelece elementos para classificação do *asset*, como descritores e informações sobre o contexto onde o *asset* está inserido.

O elemento *context* do elemento *classification* é um pouco mais complexo e merece um pouco mais de atenção. Este elemento pode nos dar informações importantes sobre o ativo em questão. A estrutura deste elemento é mostrada na Figura 4.4. 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 sobre o ativo.



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

Além destes elementos, estamos interessados em alguns atributos do elemento raiz *asset*, os quais 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, vamos incorporar no Archiva funcionalidades relativas ao RAS. Primeiramente, introduzimos alguns conceitos em relação à ferramenta Archiva e ao gerenciador Maven que contribuíram para a nossa decisão de utilizar estas ferramentas. Dando seqüência, apresentaremos a proposta de implementação que se subdivide em 3 partes: o armazenamento dos artefatos RAS no Archiva, sua apresentação e busca através de um navegador pela interface do Archiva e, por fim, o suporte aos mecanismos de busca e recuperação definidos no RAS. A apresentação dos resultados na interface do Archiva não é um requisito para o RAS. Entretanto, concluímos que já que estaremos utilizando a ferramenta, seria interessante mantermos a harmonia entre o conteúdo apresentado pela ferramenta.

### 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 componentes 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 a varredura do repositório. Neste caso, para cada artefato encontrado no repositório, cada consumidor disponível o processa. A tarefa principal deste tipo de consumidor é obter informações a respeito do artefato e garantir que suas informações sejam inseridas corretamente no indexador.

Analogamente, o segundo tipo consome ou processa artefatos durante a varredura 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 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 de um 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 e sua correlação com o arquivo descritor do RAS. 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 unicamente:

* **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 4.8.

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 .: 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 4.5. 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*.



Figura .: Tela de envio de artefato do Archiva

No nosso caso, algumas dessas informações estão presentes no arquivo descritor do artefato RAS e 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.

#### 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 4.6 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*. No diagrama da Figura 4.7 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 4.5. 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 “Identificador do Ativo” 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.



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

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 4.8). 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.

#### 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 opção *Find Artifact* da interface com o usuário 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 4.8. 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 varredura de base de dados. Na Figura 4.7, 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 4.7, 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 Tabela 4.1, 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.



Figura .: Tela de visualização de artefato JUnit do Archiva

Na Figura 4.8 podemos visualizar o artefato Java 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. Na verdade, a interface lê as informações do descritor da base de dados. Este descritor é o pom.xml, como abordado anteriormente. Assim, a Figura 4.8 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.

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 4.9



Figura .: Navegação pelos artefatos do repositório

Outro tipo de busca onde o usuário informa palavras chaves também é possível através da opção Search. Esta, entretanto, é explicada mais adiante no texto.

Quando o usuário seleciona uma versão, a classe **ShowArtifactAction** do Archiva é ativada e busca as informações relativas ao modelo de projeto do artefato. A Figura 4.10 mostra um diagrama que representa a relação existente entre os participantes deste processo.

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.



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

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 condicionalmente à existência de informação POM sobre o artefato. Quando esta não existe, devemos procurar pelas informações RAS do artefato. 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

Recuperação e Pesquisa de artefatos são funções que o Archiva já possui. No que diz respeito à recuperação de artefatos, podemos dizer que nenhuma alteração no Archiva é necessária, pois a URL fornecida já é tratada e devolve o artefato em si. Portanto, apenas precisamos nos preocupar com a pesquisa. Para esta, precisamos processar as requisições conforme a seção 9 da especificação RAS.

São duas as operações de pesquisa a serem inseridas no Archiva. A primeira delas, *SearchByKeyword*, é responsável por exibir uma lista de artefatos RAS disponíveis recebendo como entradas palavras-chave. A segunda, *SearchByLogicalPath*, deve retornar os artefatos disponíveis através de um caminho lógico para os arquivos. 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/]

#### A Pesquisa no Archiva

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.

A especificação RAS não define uma formatação para o retorno dos resultados, apenas o que eles devem conter. Decidimos representar os dados de retorno usando XML, já que este padrão é largamente aceito e independente de plataforma, muito comum para especificação de descritores.

O RAS define três tipos de descritores: Descritor de Ativo de Repositório (Repository Asset Descriptor), Descritor de Pasta de Repositório (Repository Folder Descriptor) e Descritor de Dados de Repositório (Repository Data Descriptor). Este último é, na verdade, um invólucro que pode conter tanto o primeiro quanto o segundo. A representação destes elementos em XML, bem como o formato dos resultados apresentados pelos métodos de pesquisa encontram-se no apêndice A.

#### Procura por Palavra-Chave



Figura .: Diagrama Simplificado para operação Search

O diagrama da Figura 4.11 mostra a relação entre as classes envolvidas no processo de pesquisa de artefatos. Neste diagrama, identificamos algumas relações importantes para a busca das informações de pesquisa. A primeira delas é a relação existente entre a interface com o usuário e o atributo **q** da classe **SearchAction**. Este atributo é preenchido com as palavras de pesquisa e é o principal atributo utilizado pelo método **quickSearch** desta classe, que dá início ao processo de pesquisa.

Agora que sabemos como as palavras-chave são adquiridas, podemos nos concentrar em como a lógica de pesquisa funciona. Esta explicação é importante para entendermos como podemos usar as próprias estruturas que o Archiva fornece para criarmos os nossos serviços de pesquisa. A lógica de pesquisa está presente na classe chamada **DefaultCrossRepositorySearch**. Esta classe usa o indexador padrão do Archiva através do método **searchTerm** que um objeto da classe **MultiSearcher** que executa a pesquisa em última instância. Os resultados são coletados e o retorno desta chamada, feito pelo método **quickSearch** de **SearchAction**, é atribuído ao atributo **results** desta classe. Este, por usa vez, é utilizado pela interface com o usuário para mostrar as informações. Este atributo contém todos os registros encontrados na pesquisa, cada um deles representado pela classe **SearchResultHit**.

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 4.11, 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. Nós podemos deduzir a informação de caminho lógico a partir dos campos já existentes (**groupId**, **artifactId** e **version**). No que diz respeito a descrição, podemos encontrá-la através do modelo de projeto. Entretanto, não há como deduzir ou calcular o índice de certeza, visto que o Archiva não possui esta informação em suas classes internas. 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. Se retomarmos a subseção 4.3.4, veremos que o processo descrito lá faz uso da classe **JdoArchivaDAO** para buscar as informações de modelo de projeto. Portanto, faremos o mesmo aqui, instanciando um objeto desta classe. a classe **JdoArchivaDAO** implementa uma *Abstract Factory* através da interface **ArchivaDAO**. Através desta estrutura temos acesso a um objeto da classe **JdoProjectModelDAO** a partir o qual obtemos uma referência para a represetação do modelo de projeto do artefato em questão ao chamar o método **getProjectModel**. Para que este encadeamento de chamadas funcione, basta utilizarmos os campos já conhecidos como parâmetros para esta última chamada. Todo este mecanismo é disparado através de uma requisição HTTP que é processada pelo *servlet* RasSearchByKeywordServlet no método *doGet*. O diagrama da Figura 4.12 mostra um esquemático para a seqüência descrita.



Figura .: Diagrama de Seqüência da operação de pesquisa por palavra-chave

#### Procura por Caminho Lógico

A procura por caminho lógico não utiliza o mesmo processo to passo anterior. Para explicar o processo de pesquisa por caminho lógico, precisamos retomar a Figura 4.9. Esta figura mostra como o usuário consegue navegar pelo repositório vendo sua real estrutura. Podemos observar uma estreita relação entre essa navegação com o objetivo da procura por caminho lógico. De fato, o que se está executando através da interface na Figura 4.9 é uma busca por caminho lógico. Entretanto, como estamos em um ambiente Maven, as regras para sua formação são bem específicas: no primeiro nível de diretório diz respeito ao *groupId*; no segundo, temos o *artifactId*; por último, a informação de versão que nos leva diretamente ao artefato. Como podemos perceber, o resultado de todas estas pesquisas poderá ser um descritor de artefato ou uma lista de descritores de pasta.

Desta maneira, não são muitas as formas de requisições que farão sentido neste serviço. Como não relaxamos a forma em que o artefato é submetido ao repositório, e por causa da estrutura de repositório Maven descrita acima, as únicas formas de requisições que fazem sentido na busca por caminho lógico são aquelas da Tabela 4.3.

Tabela .: Requisições RAS que fazem sentido para o contexto do Archiva

| Requisição | Significado |
| --- | --- |
| /SearchByLogicalPath?path=/ | Lista todos os groupIds existentes no repositório |
| /SearchByLogicalPath?path=/groupId | Lista todos os artifactIds existentes para o dado groupId |
| /SearchByLogicalPath?path=/groupId/artifactId | Lista todas as versões existentes para o artefato que tem groupId e artifactId |
| /SearchByLogicalPath?path=/groupId/artifactId/version | Retorna informações sobre o artefato, se ele existir. |

Tendo essas observações em mente, podemos fazer uso de um mecanismo semelhante ao exposto na subseção 4.3.4. A classe **BrowseAction** é responsável por construir as informações da Figura 4.9. Esta classe utiliza um elemento que já é de nosso conhecimento: a classe **DefaultRepositoryBrowsing**. A seguir descreveremos alguns métodos existentes na **DefaultRepositoryBrowsing** que serão responsáveis por buscar as informações necessárias para as requisições da Tabela 4.3.

* **getRoot**. Este método busca todas as pastas existentes na raiz do repositório, ou seja, todos os *Group Id* existentes no repositório.
* **selectGroupId**. Este método recebe um parâmetro que referencia um *Group Id* específico e retorna todos os *Artifact Id* existentes para este *Group Id*.
* **selectArtifactId**. Este método recebe dois parâmetros que referenciam, respectivamente, o *Group Id* e o *Artifact Id* e retorna todas as versões de um artefato *Group Id* e *Artifact Id.*
* **selectVersion**. Este método já foi abordado anteriormente quando falamos sobre a parte de visualização do artefato. Ele retorna as informações de metadados do artefato, dado que recebeu como parâmetro os respectivos *Group Id*, *Artifact Id* e versão.

Utilizando o atributo **dao**, estes métodos executam pesquisa na base de dados indicando restrições para cada um dos casos. O diagrama para esta relação já foi visto anteriormente. Entretanto, ele não apresentava a classe **BrowsingResults**. Portanto, reapresentamos parcialmente o diagrama da Figura 4.10 salientando os métodos de pesquisa e a classe **BrowsingResults**.



Figura .: DefaultRepositoryBrowsing revisado

Na implementação do serviço podemos identificar já de antemão se o resultado da pesquisa será uma lista de descritores de pasta ou o descritor de um artefato. Se o parâmetro **path** contiver os elementos **groupId**, **artifactId** e **version**, então devemos utilizar o método **selectVersion** que retorna os metadados do artefato e, portanto, o resultado desta consulta serão as informações do artefato. Adicionalmente, verificamos se o artefato segue o padrão RAS. De outra forma, utilizamos os métodos **getRoot**, **selectGroupId** ou **selectArtifactId** (baseado no número de níveis informados através do parâmetro **path**) para obter uma instância da classe **BrowsingResults** com os resultados necessários. Se a consulta possui o primeiro nível apenas (ou seja, “/”), utilizamos o método **getRoot** e buscaremos pelo atributo **groupIds** de **BrowsingResults**. Se a consulta possui apenas dois níveis (isto é, “/groupId/”), então utilizamos o método **selectGroupId**, e nossos resultados são baseados no atributo **artifacts** de **BrowsingResults**. Por último, se a consulta possui os três níveis (na forma “/groupdId/artifactId/”), então podemos utilizar o método **selectArtifactId** e buscar pelos resultados no atributo **versions** de **BrowsingResults**. Todas essas iterações podem ser observadas no diagrama da Figura 4.14.



Figura .: Relação das possíveis iterações entre o serviço de busca por caminho lógico RAS e a classe DefaultRepositoryBrowsing.

# 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>