Desenvolvimento de sistemas baseados em artefatos de conhecimento

Gustavo Salazar Torres

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Área de Concentração: Ciência da Computação
Orientador: Prof. Dr. Flávio Soares Corrêa da Silva

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES e da Microsoft Research

São Paulo, Junho de 2008

Desenvolvimento de sistemas baseados em artefatos de conhecimento

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Gustavo Salazar Torres
e aprovada pela Comissão Julgadora.

Banca Examinadora:

- Prof. Dr. Flávio Soares Corrêa da Silva (orientador) - IME-USP.
- Profa. Dra. Ana Cristina Vieira de Melo - IME-USP.
- Prof. Dr. Marco Túlio Carvalho de Andrade - EP-USP.

Agradecimentos

Em primeiro lugar, quero agradecer a Deus por todas as pessoas que conheci ao longo deste trabalho.

Quero agradecer também ao meu orientador, Prof. Dr. Flávio Soares Corrêa da Silva, pela paciência, apoio e liberdade neste projeto de pesquisa, aprendi bastante sob a sua supervisão, em especial a saber usar as minhas forças para alcançar um só objetivo.

Quero agradecer também à banca, Profa. Dra. Ana Vieira de Melo e ao Prof. Dr. Marco Túlio Carvalho de Andrade por ler este texto em tempo tão curto.

Quero agradecer também a minha companheira, Hilda, sem o seu apoio e carinho não teria chegado até hojé, e Adriana, alguém que conhecimos e mudou a nossa vida.

A minha familia por suas constantes palavras de coragem, sempre me motivando a seguir em frente diante das dificuldades, em especial aos meus pais: Nora e José, vocês são um exemplo de dedicação e esforço que espero algum dia alcançar.

Aos meus amigos da faculdade: Antônio, Cassio, Christian Paz, Crhistian Noriega, Cristian Bayes, David, Ellen, Fabio, Jesus, Karina, Marcelo, Paulo, Ricardo, Rodrigo, Sandro, muito obrigado por sua companhia e ajuda neste trabalho.

Aos meus amigos do movimento Comunhão e Libertação: Ana, Gê, Alair, Mônica, Tatiana, Valeria, Romero, Fernando, Dalton, Olivio, Aline, vocês me ensinaram (e ensinam ainda) a enxergar a realidade em todos os seus fatores, muito obrigado.

A cultura, a história, e a convivência humana fundam-se sobre esse tipo de conhecimento que se chama fé, conhecimento por fé, conhecimento indireto, conhecimento de uma realidade através da mediação de uma testemunha.

Mons. Luigi Giusanni (É possível viver assim? pág.26)

Resumo

A noção de artefato de conhecimento tem ganhado muita popularidade nas áreas de gestão de conhecimento e, mais recentemente, em sistemas baseados em conhecimento. O objetivo principal deste trabalho é duplo. Primeiro propomos um método ágil para o projeto e implementação de sistemas baseados em conhecimento com base em artefatos de conhecimento. Observamos que sistemas construídos de acordo com este método podem efetivamente implementar o fluxo de conhecimento entre diferentes comunidades de prática. O nosso método foi desenvolvido de baixo para cima, i.e. temos desenvolvido sistemas concretos baseados na noção abstrata de artefato de conhecimento e sintetizado o nosso método baseado nas reflexões sobre as nossas experiências construindo estes sistemas. Segundo, propomos um modelo lógico para artefactos de conhecimento usando linguagens da Web Semântica, e implementamos um plugin para a ferramenta Protègè com o objetivo de ajudar no desenvolvimento deste tipo de Sistemas baseados em Conhecimento.

Palavras-chave: Artefato de Conhecimento, Sistemas baseados em Conhecimento, Ontologias, Gestao de Conhecimento, Métodos Ágeis de Engenharia de Software.

Abstract

The notion of knowledge artifact has rapidly gained popularity in the fields of general knowledge management and more recently knowledge-based systems. The main goals of this work are twofold. First we propose an agile method for the design and implementation of knowledge-based systems founded on knowledge artifacts. We have observed that the systems built according to this method can be effective to convey the flow of knowledge between different communities of practice. Our method has been developed from the ground up, i.e. we have built some concrete systems based on the abstract notion of knowledge artifact and synthesized our method based on reflections upon our experiences building these systems. Secondly, we have implemented a knowledge model for knowledge artifacts and implemented a plugin in Protègè to help develop this kind of knowledge-based systems.

Keywords: Knowledge Artifacts, Knowledge Based Systems, Ontologies, Knowledge Management, Agile Software Development.

Sumário

Lista de Figuras

Capítulo 1

Introdução

A motivação para iniciar este projeto começou com o desenvolvimento de um protótipo para a uma empresa de manufatura de grande porte no Brasil. Esse protótipo tinha por objetivo mostrar a viabilidade técnica de implementar um sistema baseado em artefatos de conhecimento (SBAC) e as possíveis vantagens que este traria para a manutenção e reutilização de sistemas para projeto e manutenção de dispositivos complexos. Artefatos de conhecimento (AC) são uma proposta recente na área de representação de conhecimento [9], [17]. Segundo Holsapple e Joshi [31], um AC "é um objeto que contém e carrega uma representação útil de conhecimento".
As conclusões desses trabalhos sugeriram estudar aspectos metodológicos da identificação e utilização de AC no contexto bem definido das Comunidades de Prática (CDP). Segundo Wenger [53], as Comunidades de Prática são "grupos de pessoas informalmente unidas por afinidades conjuntas".

Por outro lado, na área de Engenharia de Conhecimento, recentemente foram propostos métodos ágeis para o desenvolvimento de Sistemas baseados em conhecimento (SBC) ([32], [7]). Em geral, estes métodos tentam extender os métodos de desenvolvimento ágil de sistemas de software para o desenvolvimento de Sistemas baseados em conhecimento tentando ao mesmo tempo incluir práticas que permitam modelar conhecimento de forma eficiente.

O problema a ser resolvido neste trabalho é extender esses métodos ágeis de Engenharia de conhecimento para o desenvolvimento de SBAC. A extensão deverá incluir a experiência desenvolvendo este tipo de sistemas assim como um estudo teórico das noções de Artefato de conhecimento, Comunidade de Prática e Métodos ágeis de Engenharia de conhecimento.

Podemos dividir esse trabalho nos seguintes objetivos:

1. Propor uma métodо ágil de engenharia de conhecimento para o desenvolvimento de SBAC com base no estudo das CDP [52], os AC e os métodos ágeis de engenharia de conhecimento existentes na literatura.

2. Desenvolver software para ajudar na criação, desenvolvimento e utilização dos AC:

   - Uma ferramenta de software que ajudará no projeto de modelos baseados em AC. Algumas noções dadas em [54] serão seguidas no desenvolvimento desta ferramenta.

No capítulo 2 descrevemos o conceito de AC e sua relação com sistemas baseados em conhecimento (SBC). No capítulo 3 são apresentados os aspectos mais relevantes dos métodos ágeis XP.K e RapidOWL.

No capítulo 4 descreve-se a nossa proposta metodológica com base no estudo anterior. No capítulo 5 é descrita uma proposta de modelo lógico em OWL para modelos baseados em AC. No mesmo capítulo explicamos o desenvolvimento de uma ferramenta para ajudar na criação e apresentação desses modelos. A ferramenta foi implementada como um plugin em Protègè. No anexo A mostramos a utilização dessa ferramenta tomando como exemplo o caso de estudo do capítulo 2.

Finalmente, no capítulo 6 apresentamos algumas conclusões e trabalhos futuros.

Capítulo 2

Artefatos Computacionais de Conhecimento

Em [30] encontramos que um artefato é um objeto produzido intencionalmente para um propósito específico. Além disso, de acordo com [36], um artefato é o resultado de uma atividade humana disciplinada, seguindo regras e baseado em treinamento e experiência. Como conseqüência, cada artefato tem um autor e um propósito.

Artefatos podem ser avaliados com base em como suas características reais são compatíveis com as características pretendidas por seus autores e os propósitos para os quais foram construídos. Dado um propósito P, um autor A projeta um artefato e obtém uma invenção, uma idéia ou um projeto que descreve o artefato em um nível abstrato. Vamos nos referir a este objeto, que é resultado do trabalho inteletual do autor, usando o símbolo I. Finalmente, um artefato final R que implementa I é usado para o propósito P. Deve-se notar que o "propósito" que gerou o projeto de I e a implementação do artefato R pode diferir do propósito do usuário de um artefato.

O artefato R descrito por I é um objeto final. O autor gostaria que R tivesse as características pretendidas para ser compatível com o propósito original P. A descrição I de um artefato é o resultado de um processo de *design*[1]. Existe uma representação formal e abstrata de um objeto final R. O artefato inventado necessita ser implementado, com o objetivo de produzir o artefato final R que é utilizável.

Como mostrado na Figura 2.1, P, I e R estão relacionados através do *design*, *implementação* e *utilização*. Um conjunto bem sucedido de relações é aquele em que o design leva a um I que é perfeitamente compatível com um propósito P, cuja implementação leva a um R perfeitamente

[1]Não existe em português uma tradução exata para o termo em inglês "design". Por esse motivo, utilizaremos neste texto o termo em inglês

compatível com I, tal que a utilização de R é também perfeitamente compatível com P.

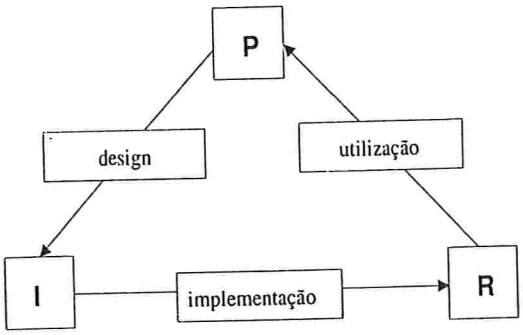

Figura 2.1: Design, Implementação e Utilização de um Artefato (adaptado de [25])

Um ponto interessante a ser notado é que as flechas na Figura 2.1 são relações de tipo 1-N: um propósito P pode levar a uma variedade de projetos I_1,I_2, etc. Por exemplo, o propósito de escrever um artigo pode levar ao projeto de uma caneta, um lápis, alguma classe de pincel, etc. ou a um projeto mais detalhado, como "projeto XYZ de uma caneta esferográfica". Analogamente, o projeto I pode levar a uma variedade de implementações R_1, R_2, etc. Por exemplo, a implementação de uma caneta esferográfica (ou de um "modelo XYZ de caneta esferografica" mais específico) pode levar a objetos reais, cada um com sua ponta específica, forma e recarga de tinta. Finalmente, um artefato real R pode levar a uma variedade de propósitos P_1, P_2, etc., inclusive propósitos não levados em conta inicialmente durante o design e a implementação. Um exemplo notável são os tocadiscos, inicialmente usados para o propósito de reproduzir música previamente gravada e agora usados para criar peças originais de musica, p.ex. no hip-hop.

Um artefato é aqui caracterizado por uma tripla (I, R, P), onde o projeto I é implementado como un objeto real R que é usado para um propósito P (o qual, como foi mencionado anteriormente, não é necessariamente o propósito P_0 que levou ao projeto de I). Os elementos separados I, R, e P, podem ser colocados junto em uma tripla (I, R, P) que é o "artefato em contexto" do artefato R.

Conforme apontado em [20], definir conhecimento não é uma tarefa fácil ou já resolvida. Algumas caracterizações úteis de conhecimento são:

- Um componente na hierarquia Dados-Informação-Conhecimento ([21], [50], [51]): dados sao considerados fatos observáveis e codificáveis; informação são dados providos de significado;

conhecimento é informação acionável, i.e. informação que provê agentes com os recursos necessarios para realizar ações.

- A posição filosófica [22]: conhecimento é uma crença verdadeira justificada. Esta caracterização, que se remonta a Platão, é uma caracterização precisa, mas se baseia em outros três conceitos complexos: crença, verdade e justificabilidade.

- A posição baseada em agentes [20]: conhecimento é a capacidade de um agente mudar o mundo. O mundo muda sempre que o valor de verdade de uma sentença que descreve uma das suas características precisa ser atualizado. Quando tal mudança ocorre como resultado da ação propositada de um agente, o agente necessariamente tem conhecimento.

Estas caracterizações são complementares entre si. Empregamos as três ao longo deste trabalho.

Dado um propósito P, um agente pode projetar um artefato I "feito de" conhecimento, i.e. dados interpretáveis e codificáveis tais que, quando apresentados para qualquer outro agente, dá a este agente a capacidade de realizar alguma ação que pode mudar o mundo. O artefato projetado I é tal que, quando incorporado em uma implementação R, pode ser efetivamente usado para mudar o mundo.

Como exemplo ilustrativo, podemos considerar o propósito P de preparar um bolo de chocolate. Um agente (i.e. o confeiteiro neste caso) pode inventar um artefato I, ou seja uma receita provendo os passos para preparar um bolo. O artefato projetado I é generico: não especifica que o bolo será de chocolate. Consideramos uma classe genérica de bolos que podem ser provisoriamente chamados de "bolos para acompanhar chá", os quais compartilham os mesmos passos gerais e ingredientes para ser preparados. A receita R é a receita concreta obtida deste processo e é derivada a partir do artefato projetado I. A receita obtida pode ser usada para outros propósitos, por exemplo de forma que um indivíduo possa preparar um bolo de chocolate (propósito P), ou que um estudante em uma escola de culinária possa completar sua lição de casa (propósito P').

Encontramos muitas definições de AC na literatura. Estas definições podem ser classificadas a grosso modo nos seguintes quatro grupos:

1. um AC é um artefato que representa conhecimento;

2. um AC é um artefato que representa uma codificação de conhecimento;

3. um AC é um artefato cuja criação requer conhecimento especializado;

4. um AC é um artefato "feito de" conhecimento.

Uma das definições mais frequentemente usadas de um AC se deve a Holsapple e Joshi. Eles descrevem AC como "objetos que transportam e contêm uma representação usável de conhecimento" [31]. Se tomamos a definição 2 acima junto com a definição de Holsapple e Joshi, temos codificações de conhecimento usáveis -i.e. executáveis - que podem ser implementadas como programas de computador, escritos em uma linguagem de programação como C, Java, ou linguagens declarativas de modelagem como XML, OWL ou SQL.

A utilidade de um AC resultado da perspectiva a partir da qual ele é abordado. Existem pelo menos quatro perspectivas que podem ser consideradas:

- A *Perspectiva de Gerenciamento de Conhecimento* que visa desenvolver sistemas para criar, armazenar e gerenciar AC. De acordo com esta perspectiva, um AC é um objeto que contém alguma representação de conhecimento (documentos, arquivos, etc.), e o seu gerenciamento pode acrescentar valor à organização através da acessibilidade melhorada aos seus recursos de conhecimento. Com base nesta perspectiva, AC são veículos para compartilhar conhecimento.

- A *Perspectiva de Suporte a Comunidades* que visa criar locais e objetos nos quais os processos comunitários (e.g. negociação) são apoiados. Estes processos geram objetos compartilhados (os AC), tais como emails ou o conteúdo de um blog. De acordo como esta perspectiva, os AC fazem ainda o papel de transportadores de conhecimento compartilhado, além de contribuintes para a evolução da comunidade mediante o suporte dos seus processos naturais de aprendizagem.

- A *Perspectiva Computacional*, que considera AC como representações de conhecimento armazenadas para ser usadas em computações dentro de um método particular de resolução de problemas em um sistema computacional;

- A *Perspectiva de Inteligência Artificial*, que tem por objetivo separar o conhecimento da sua utilização. O AC deve persistir inclusive sem um sistema para o seu gerenciamento. Além disso, a sua identificação em uma organização pode ser uma guia útil para o desenvolvimento de SBC mediante a aplicação de uma metodologia baseada em AC.

Neste trabalho vamos nos focar na perspectiva de Inteligência Artificial e considerar um AC como um artefato cujo constituinte primário é conhecimento.

Um exemplo mais concreto de um AC é o processo de decisão que um engenheiro usa no reparo de dispositivos complexos (e.g. um motor de carro). Por exemplo, dado um propósito específico P de reparar um pistão com uma trinca, o agente (que é um engenheiro neste caso) deve tentar reparar o pistão. Para poder fazer isto, o engenheiro obtém através da aplicação de um procedimento de reparo um pistão reparado com as mesmas características mecânicas de antes de receber o dano. Neste caso, o modelo de conhecimento envolvido no reparo de um pistão com dano é o AC da comunidade de engenheiros.

O conceito de AC é interessante porque caracteriza oportunidades para reutilizar componentes das classes P, I, e R para montar diferentes artefatos, que podem ser úteis para o compartilhamento de conhecimento com diferentes objetos de fronteira entre diversas Comunidades de Prática (CDP) [52]. Um AC que se posiciona entre duas ou mais CDP e atua como uma ponte semântica entre essas CDP é denominado objeto de fronteira, pois ele se torna o ponto de contato entre as CDP [8].

Já que um AC é essencialmente um artefato, é possível descrevê-lo através de triplas (P, I, R), incluindo-o no ciclo *design-implementação-utilização*. Considerando uma CDP específica, o propósito P é a meta da comunidade (e.g. o projeto de um produto ou de um processo de manufatura). De acordo com o ciclo proposto, I é o resultado de uma atividade de design: neste caso, I não é um objeto simples e sim a união de elementos da CDP (e.g. papéis formais ou tácitos, estratégias e hábitos dos participantes da CDP) que diretamente influenciam o processo de tomada de decisões específico da CDP. Para completar a tripla, R é o modelo de conhecimento envolvido neste processo de tomada de decisões, compartilhado entre a comunidade.

Na seção seguinte daremos um exemplo de um AC, que é o resultado da implementação de um SBC. O exemplo fará uso de termos usados no reparo de motores de carros de alta performance (Ferrari, etc). A experiência na implementação deste SBC, junto com a do projeto P-Truck [9], servem como base empírica para a formulação de um metodo ágil para projetar e implementar SBC baseados em AC [25].

2.1 Caso de Estudo

O AC *Maintenance and Repair* foi implementado em uma grande companhia de manufatura para apoio ao diagnóstico e reparo de dispositivos eletro-mecânicos complexos. Nos parágrafos a seguir vamos descrever o domínio de aplicação e depois serão dadas regras BNF que descrevem este artefato.

2.1.1 Descrição do Domínio

Maintenance and Repair tem a ver com a tarefa de diagnóstico e reparo de uma classe de dispositivos complexos nos quais danos podem ser identificados em alguns dos seus componentes. O propósito é restabelecer as suas propriedades originais de forma a reduzir os custos de manutenção. O AC é usado principalmente em ambientes de manutenção onde um modelo abstrato é usado para descrever o dispositivo (e.g. um Modelo de Elementos Finitos), dado que seria muito custoso desenvolver dispositivos reais e testar fisicamente cenários com muitos danos.

O processo de diagnóstico e reparo começa com a aplicação de danos sobre a estrutura do dispositivo. A descrição dos danos é detalhada, já que a estrutura pode aceitar danos nos seus componentes mais finos.

Os danos são classificados de acordo com os seus atributos (e.g. profundidade, localização) através da estrutura. De acordo com esta classificação, as partes danificadas são testadas usando os modelos formais para definir o seu status. Finalmente, procedimentos de reparo são aplicados para devolver as propriedades originais à estrutura.

Para mostrar a estrutura do AC *Maintenance and Repair*, usaremos o domínio de reparo de motores de carro.

2.1.2 Conhecimento Ontológico

A gramática descrevendo a estrutura ontológica inclui, no caso de um carro, o motor e suas peças:

```
<device> ::= "Nissan350ZEngine" | "FerrariEngine" | ...
<device description> ::= <device> " has " <part>
<part> ::=
    <cylinder> | <piston> | <spark plug> |
    <valve> | <cylinder bank>
```

Por exemplo podemos descrever "FerrariEngine" com a sentença "FerrariEngine has bank1" para indicar que "FerrariEngine" tem um banco de cilindros (cylinder bank?) "bank1"

Já que o propósito do AC Maintenance and Repair é reparar componentes finos, é necessario ir mais a fundo na descrição do dispositivo:

```
<cylinder arrange> ::= "v" | "flat" | "inline"
```

```
<cylinder bank> ::= "bank1" | "bank2"

<cylinder bank description> ::=
    <cylinder bank> " has " <cylinder arrange> " arrange" |
    <cylinder bank> " has " <number> "cylinders" |
    <cylinder bank> " is composed by " <cylinder>

<cylinder material> ::= "aluminum" | ...

<cylinder> ::= "cylinder1" | "cylinder2" | ...

<cylinder description> ::=
    <cylinder> " contains " <piston> |
    <cylinder> "has depth" <real number> |
    <cylinder> " has " <cylinder material>
```

Continuando com o exemplo anterior, poderíamos descrever o banco de cilindros "bank1" com as sentenças "bank1 has V arrange", "bank1 has 10 cylinders", "bank1 is composed of cylinder1" e "cylinder contains piston1". Descrições mais profundas podem incluir o pistão, o seu material (e.g. aluminio), a espessura de elementos mais finos como a cabeça e o revestimento do pistão:

```
<piston> ::= "piston1" | "piston2" | ...

<piston descripton> ::=
    <piston> " has " <piston material> " material" |
    "(" <piston> "," <piston component> ")"

<system> ::= "mm" | "cm"

<piston component> ::= "piston1head" | "piston1skirt" | ...

<piston material> ::= "aluminum" | "ceramic-aluminum"

<piston component type> ::= "piston head" | "piston skirt"

<piston component description> ::=
    "(" <piston component> "," <piston component type>, <piston component property> "," <amount> <system> ")"

<piston component property> ::= "thickness"

<damageable component> ::= <piston component>

<damageable component type> ::= <piston component type>

<damageable component property> ::= <piston component property>
```

Podemos descrever a cabeça de 'piston1" como "(piston1,piston1head)" para dizer que "piston1" está composto por "piston1head". Na sentença "(piston1,piston head, thickness, 5mm)" descrevemos profundamente "piston1head" dizendo que é uma peça do tipo cabeça de pistão com uma espessura de 5 mm.

2.1.3 Conhecimento epistemológico

É preciso descrever agora quais classes de danos podem ser aplicadas a este dispositivo e como estas estão relacionadas com os elementos do motor de carro. Cabe ressaltar que nem todas as peças podem ter danos relacionados. A idéia básica é ter danos categorizados por status e classe. O status diz respeito a como aplicar o reparo (se é que pode ser reparado) e a classe informa qual o tipo de dano. Esta ultima propriedade pode também indicar se uma peça em particular com dano pode ter um reparo permanente ou temporario. O propósito deste AC é devolver as propriedades iniciais das peças com dano, de forma que o dispositivo inteiro possa continuar funcionando:

```
<repair> ::= "repair1" | "repair2" | ...
<repair type> ::= "temporal" | "permanent" | ...
<repair description> ::=
    "(" <repair> "," <repair type> "," <damage> ")"
<damage> ::= "damage1" | "damage2" | ...
<damage type> ::= <mechanical> | <termo-mechanical>
<mechanical> ::= "crack" | "hole" | ...
<damage status> ::= "admissible" | ...
<damage description> ::=
    <damage> " is a " <damage type> |
    "(" <damage> "," <damageable component> "," <damageable component property> "," <amount> <system> ")"
<damage classified> ::= "(" <damage> "," <damage status>")"
<damage limit> ::=
    "(" <damage status> "," <damageable component type> "," <damageable component property> "," <amount> "%)"
```

Podemos supor que "piston1" tem um dano na cabeça descrito pela sentença "(damage1," "piston1head,thickness,1.5mm)".
Cada dano reduz alguma propriedade do componente, neste caso a sua espessura (e.g. em uma vela poderia diminuir a resistência ao calor). Poderiamos refinar a nossa descrição com a sentença "damage1 is a crack", dando a entender que a redução de espessura no pistão foi causada por uma trinca.

2.1.4 Suporte baseado em Artefato de Conhecimento

O Método de Resolução de Problemas na Figura 2.2 determina o status do dano (e.g. status admissível significa que o dano pode ser permitido de ficar no componente afetado) e aplica o procedimento de reparo para devolver as propriedades iniciais dessa peça.

Seguindo o nosso exemplo, precisamos determinar o status de "damage1". Para este propósito usamos a regra gramatical "(damage limit)" para estabelecer a quantidade máxima de redução na propriedade de um componente. Se a quantidade de dano é maior que o máximo permitido, então é classificada como reparo, caso contrário é um dano admissível ou alguma outra classe de status.

No nosso exemplo, poderíamos ter uma regra limite de 20% da espessura original da cabeça do pistão para danos admissíveis "(admissible,piston head, thickness, 20%)", assim "damage1" não é um dano admissível e exige um reparo permanente indicado pela regra "(repair1,permanent,damage1)". Antes de aplicar o reparo, a parte danificada é submetida a testes formais simulados para estabelecer se existe um procedimento de reparo para esta classe de dano. Geralmente, no caso das cabeças de pistão danificadas, o pistão inteiro é substituido.

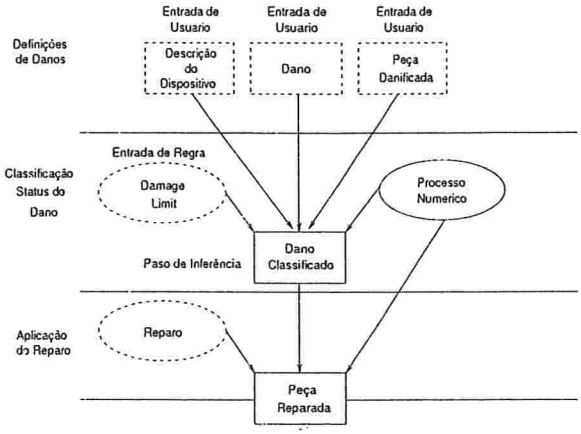

Figura 2.2: Método de Resolução de Problemas de Maintenance and Repair

2.2 Sistemas Baseados em Conhecimento

Um SBC é um sistema computacional que representa e usa conhecimento para levar a cabo uma tarefa [45]. A disseminação de aplicações deste tipo de sistema levou a que o termo mais geral "sistemas baseados em conhecimento" fosse preferido por algumas pessoas em vez de "Sistema Especialista", porque foca a atenção no conhecimento que os sistemas transportam, em vez de responder à pergunta de se o conhecimento é especializado ou não.

2.2.1 Componentes de um SBC

A arquitetura clássica de um SBC inclui uma base de conhecimento, duas interfaces de usuário, e um subsistema de inferência ou de busca. A base de conhecimento é um repositório (geralmente uma base de dados) onde o conhecimento usado pelo sistema é armazenado (Figura 2.3). Periodicamente, a base de conhecimento precisa ser modificada para refletir as mudanças no modelo do domínio [45]. Uma interface de usuário é responsável por interagir com os usuários do sistema na resolução de problemas no seu domínio. Tarefas como fazer perguntas sobre o problema sendo resolvido, resolver consultas e mostrar resultados são típicas deste componente. A segunda interface de usuário é a

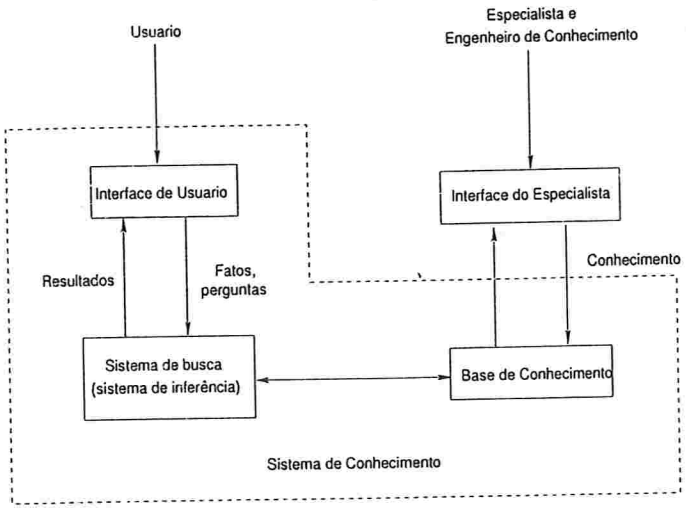

Figura 2.3: Arquitetura Clássica de Sistemas Baseados em Conhecimento(adaptado de [45])

interface do especialista. Esta é a interface pela qual o conhecimento é inserido no sistema. A interface do especialista é usada pela equipe de elicitação de conhecimento formada por um especialista e um engenheiro de conhecimento. A interface provê também acesso para atualizar, testar, e depurar a base de conhecimento, incluindo ferramentas para examinar o conteúdo da base de conhecimento. Isto quer dizer que os usuários não podem usar este tipo de interface porque fazer uma mudança na base de conhecimento por alguém não treinado pode gerar efeitos não previstos nem desejados.
O sistema de inferência é a parte do sistema que "raciocina" para chegar a uma solução a partir do problema. Utiliza um Método de resolução de problemas específico para guiar a obtenção da solução de um problema.

2.2.2 Ciclo de vida de um SBC

Freqüentemente, o ciclo de vida de um SBC é esquematizado nas fases indicadas na Figura 2.4: *aquisição de conhecimento* (ou elicitação) na qual o domínio de aplicação é analisado usando as fontes de conhecimento disponíveis e através de entrevistas com os especialistas; *engenharia de conhecimento* que busca modelar e representar o conhecimento capturado em uma forma adequada para a fase de codificação;*desenvolvimento de software*, que se preocupa com a construção efetiva da aplicação. Finalmente, depois de todas estas fases, o *uso* da aplicação obtida começa.

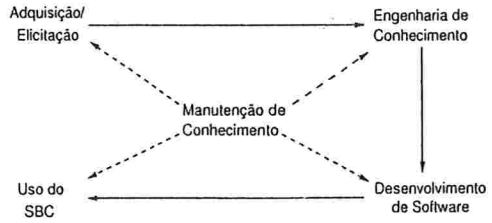

Figura 2.4: A Manutenção de Conhecimento esta relacionada com cada fase do ciclo de vida. (Adaptado de [17])

2.2.3 Papéis e Artefatos no desenvolvimento de um SBC

Com base nesta definição de SBC podemos identificar os papéis e artefatos envolvidos no processo de desenvolvimento de um SBC:

Agentes do Domínio Eles provêem o conhecimento do domínio diretamente ou indicando outras fontes de conhecimento como livros ou manuais. Existem dois tipos de agentes: os *Especialistas*, que são todas aquelas pessoas que tomam parte na fase de aquisição do conhecimento e possuem o conhecimento requerido no processo de resolução do problema, e os *Usuários*, que são os que realmente utilizam o sistema.

Engenheiros de Conhecimento São a ligação principal entre os *Especialistas* e o aspecto técnico do sistema.

Desenvolvedores de ferramentas Constroem ou adaptam ferramentas que permitem aos *Especialistas* e *Engenheiros de Conhecimento*, modelar o conhecimento em uma linguagem de modelagem específica. Se são utilizadas ferramentas padrão, este papel não aparece.

Desenvolvedores de sistemas Desenvolvem todo o sistema mediante a integração dos módulos de conhecimento com outros componentes.

Durante a atividade de *Manutenção de Conhecimento*, podem acontecer sobreposições entre as responsabilidades do *Especialista* e do *Engenheiro de Conhecimento*. Isto acontece quando o especialista que contém o conhecimento sabe qual é o conjunto de modificações importantes que podem ser feitas na base de conhecimento, de maneira a mantê-lo atualizado, e age diretamente no sistema para efetivar estas mudanças.

Quanto aos artefatos usados no desenvolvimento do SBC temos:

- Os *modelos* que representam o conhecimento do domínio e podem ser representado em três níveis, sendo que cada um deles descreve um modelo distinto: (1) o *modelo do especialista* descreve, na linguagem técnica usada por ele, a forma como ele enfrenta o problema. Cabe ressaltar que esta linguagem é informal; (2) o *modelo de conhecimento*, que é uma tradução do modelo anterior em uma linguagem formal (e.g. lógica de primeira ordem) ou semi-formal (e.g. redes semânticas); e (3) a *base de conhecimento*, que é obtida a partir do modelo anterior depois de passar por uma etapa de escolha de uma linguagem de programação apropriada para a sua implementação (e.g. Java, Lisp, Prolog).
- as *ferramentas de modelagem* são usadas para visualizar e editar os modelos. A definição ou escolha de um modelo tem um impacto significativo na forma como a ferramenta será ou deverá ser usada.
- Finalmente, o SBC exccutável consiste de outros componentes, tais como os algoritmos de raciocínio e as interfaces de usuário.

2.3 Sistemas baseados em Artefatos de Conhecimento

O uso de AC no ciclo de desenvolvimento de SBC obriga a introduzir algumas mudanças na sua arquitetura assim como nos modelos usados. Nesta seção vamos explicar brevemente essas mudanças propostas em [17].

Quanto aos modelos usados é introduzido o *Modelo baseado em AC* que é o resultado da aplicação de um AC na elicitação de um *modelo de conhecimento*.
O modelo baseado em AC é constituido de três componentes:

Intensional Que contém uma descrição dos elementos ontológicos e epistemológicos do domínio sendo elicitado e que são utilizados na tomada de decisões dos especialistas envolvidos. Embora este componente esteja presente em todo modelo de conhecimento, neste caso uma gramática (contendo termos e regras de composição) descreve os elementos ontológicos (e.g. no caso de estudo mostrado anteriormente temos os elementos gramaticais "⟨*device*⟩"e "⟨*part*⟩", que descrevem um dispositivo e as partes que o compõem).

Extensional Este componente é composto pelas sentenças com base nas regras dadas pela gramática definida anteriormente.

Raciocínio Aqui é descrito o Metodo de resolução de problemas usado na tomada de decisões dos especialistas. Ele tem como entradas as sentenças dadas pelo componente extensional e que depois devem ser interpretadas pelo SBC para sua avaliação usando algum motor de inferência.

A principal característica de um AC é a separação do uso do conhecimento da sua representação com a finalidade de facilitar a manutenção do modelo baseado em AC e, ao mesmo tempo, a sua utilização por sistemas computacionais (e.g. Sistemas baseados em Conhecimento). Com respeito ao aspecto de software, um dos efeitos de usar um AC no desenvolvimento de um Sistema baseado em Conhecimento (devido à separação do uso e a representação de conhecimento) é que este precisa de uma arquitetura diferente. Esta arquitetura contém 4 camadas [17] como mostra a Figura 2.5.

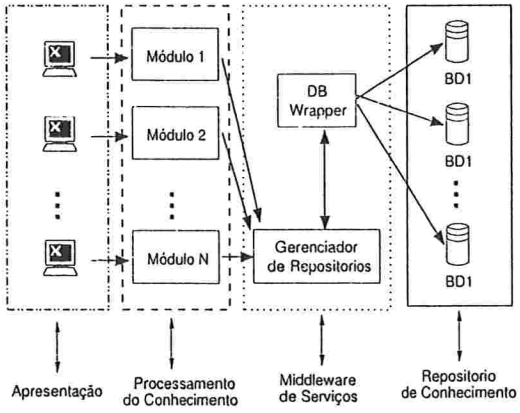

Figura 2.5: Arquitetura de Sistemas baseados em AC(adaptado de [17])

A seguir, descrevemos cada uma dessas camadas assim como os componentes que as conformam:

Repositórios de Conhecimento Elementos de software que armazenam representações de conhecimento. Geralmente implementados em Bases de Dados (e.g. MySQL).

Módulos de processamento de conhecimento Esta camada implementa o processo no qual, através de inferências baseadas no conhecimento armazenado e um método especifico de resolução de problemas, um SBC encontra uma solução qualitativa ao problema para o qual foi construído. Podem ser implementados muitos métodos de resolução de problemas dependendo da necessidade dos usuários.

Serviços Middleware Os componentes desta camada têm a responsabilidade de manter o fluxo efetivo de representações de conhecimento a partir dos repositórios nos quais estão armazenados até os outros componentes de software, e vice-versa. O componente "Gerenciador de Repositorios" se encarrega de abstrair todos os acessos aos repositorios de dados (camada de Repositorios de Conhecimento) assim como codificar as operações que outros componentes queiram fazer aos repositorios. O componente "DB Wrapper" decodifica essas operações e as converte em sentências que as bases de dados possam processar (e.g. sentências SQL).

Módulos de Apresentação Esta camada contém todos os componentes de interface com o usuário.

Capítulo 3

Métodos Ágeis de Engenharia de Conhecimento

A Engenharia de Conhecimento busca propor metodologias que permitam construir um SBC de uma forma controlada e sistemática [47].

Dentro desta área de pesquisa há muitas metodologias, ressaltando entre elas CommonKADS [42] e MIKE [2]. A intenção deste tipo de metodologias, chamadas de cascata ("waterfall"), é impor um processo relativamente rígido, que tem muita similaridade com metodologias tradicionais de Engenharia de Software, ao desenvolvimento de SBC.

Porém, essas metodologias tiveram um sucesso limitado, pois não são capazes de modelar o próprio processo de modelagem de conhecimento [32], [41] que é evolutivo e colaborativo.

Um outro conjunto de metodologias são as orientadas à modelagem de conhecimento ontológico, como On-To-Knowledge [49] e Methontology [35]. Essas metodologias utilizam também modelos de processos em cascata para identificar, implementar e garantir a manutenção de uma ontologia. Cabe ressaltar que On-To-Knowledge utiliza linguagens da web semântica como RDF [56] e OIL [24] para expressar a ontologia mediante o uso de ferramentas de edição como OntoEdit [48] e de armazenamento como Sesame [13].

Métodos ágeis de engenharia de conhecimento como XP.K [32] e RapidOWL [7], além de outros trabalhos [10], propõem abordar de forma mais efetiva as características do processo de modelagem de conhecimento baseando-se nos princípios dos métodos ágeis. Nas seções seguintes faremos uma introdução breve ao estado da arte, descrevendo as metodologias ágeis XP.K e RapidOWL.

3.1 Definição de Metodologia

De acordo com [32], uma metodologia é " um acordo de como várias pessoas irão trabalhar juntas. Ele define um processo no qual a equipe de desenvolvimento construirá modelos, incluindo o sistema executável. Estes modelos são construídos usando linguagens de modelagem junto com ferramentas adequadas. Processos, linguagens e ferramentas são baseadas em paradigmas de modelagem." A Figura 3.1 mostra o processo de construção e uso de uma metodologia, começando com os paradigmas na base da pirâmide e terminando com o uso que serve para validar os modelos e processo definidos pela metodologia.

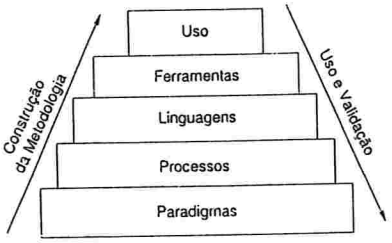

Figura 3.1: Noção de metodologia (adaptado de [32])

Esta definição é adotada pelas metodologias apresentadas nas seguintes seções. Vamos nos focar nos métodos ágeis que elas propõem pois servirão de base para a o nosso método proposto no seguinte capítulo.

3.2 XP.K

XP.K [32] é uma metodologia ágil que propõe o uso de Programação Extrema (XP) em seu modelo de processo e UML como linguagem de modelagem de conhecimento. Com base nessa linguagem são propostas ferramentas (e.g. KBeans) para gerar codigo fonte em Java usando engenheria Round-Trip. Isto permite ao usuário realizar testes automaticos sobre a base de conhecimento. As principais contribuições de XP.K [32] são:

- Uma análise do processo de modelagem de conhecimento e o seu efeito no desenvolvimento de Sistemas baseado em Conhecimento (SBCs). Com base nessa análise, é proposto um conjunto de requisitos que toda metodologia de engenharia de conhecimento deve resolver.
- A identificação de XP como base para uma metodologia que resolve esses requisitos. A mo-

dificação de algumas das suas práticas é feita para otimizá-las com respeito às demandas específicas do processo de modelagem de conhecimento.

- O KBeans [33], que é um framework de software orientado a objetos em Java que serve de meta-modelo para a modelagem de ontologias. Este framework estende os JBeans e inclui mecanismos que permitem a verificação rápida da consistência do modelo sendo elicitado.
- O KBeansShell [34], que é uma ferramenta para modelar conhecimento baseada no framework KBeans e utiliza UML como linguagem de modelagem.

3.2.1 Requisitos de Modelagem de Conhecimento

Como foi mencionado no inicio desta seção, uma das contribuições de XP.K foi analisar o processo de modelagem de conhecimento para identificar os requisitos que toda metodologia de engenharia de conhecimento deve cumprir. A continuação descrevemos as características desse processo:

- Existe muita comunicação (ou interação) entre os *Engenheiros de Conhecimento* e os *Agentes Especialistas* no processo de elicitação do modelo de uma base de conhecimento, devido ao conhecimento ser construído de forma comunitária [40].
- O conhecimento humano é iterativo e baseado em feedback [41]. Isto se traduz na modelagem dos vários modelos de conhecimento utilizados no desenvolvimento de um SBC. Existem muitas razões para um modelo mudar no tempo (e.g. os requisitos dos usuários mudam), por isso é importante testar esses modelos com dados simulados e reais para obter feedback.

Qualquer metodologia de engenharia de conhecimento deve levar em conta estas características da modelagem de conhecimento. Elas impõem requisitos sobre cada uma das camadas que compoem uma metodologia (Figura 3.1):

Processo Todo bom processo de desenvolvimento que aborde a modelagem de conhecimento deve considerar o feedback, colaboração e a mudança na sua definição. O modelo de processo deve ser iterativo e evolutivo para "...evitar o perfeccionismo incorreto e a rigidez prematura no desenvolvimento de máquinas de conhecimento (e.g. Sistemas baseados em Conhecimento)" [39, pág. 260].

Linguagem Toda linguagem de modelagem de conhecimento deve ser *fácil de aprender* e ter uma expressividade *adequada* ao domínio. Além disso, a transição entre estas linguagens de alto nível e o sistema executável deve ser fluída.

Para permitir a evolução dos modelos de conhecimento, a linguagem de modelagem deve permitir *mudanças incrementais* e tolerância a modelos incompletos. A construção de SBCs confiáveis usa linguagens de modelagem formais e não-ambíguas, o que facilita a sua verificação.

Finalmente, para poder dar suporte à reutilização, comunicação e distribuição destes modelos, a *compatibilidade* com linguagens padrão é desejável.

Ferramentas As ferramentas usadas em qualquer metodologia devem ter um design simples e ser fáceis de aprender, porque serão usadas pelos especialistas. Devem também ajudar a reduzir a transição entre os modelos e o sistema executável, assim como auxiliar na criação de modelos corretos mediante a sua constante verificação.

Finalmente, as ferramentas devem ser *agradáveis*, para motivar a equipe melhorando a comunicação.

3.2.2 Metodologia

A meta de XP.K é tornar o desenvolvimento de SBC do "mundo real" mais eficiente. Para obter uma abordagem de desenvolvimento prática, a metodologia tenta se aproximar o máximo possível de tecnologias da indústria de software. XP.K combina a Engenharia de Software e a Engenharia de Conhecimento de forma que as fraquezas de uma são cobertas pelos pontos fortes da outra. Basicamente esta metodologia é uma extensão da tecnologia orientada a objetos mediante sub-processos, elementos de linguagem, e ferramentas que cobrem os requisitos específicos da modelagem de conhecimento.

Paradigmas: XP.K cobre a maior parte dos conceitos orientados a conhecimento (e.g. ontologias) com princípios padrão de Orientação a Objetos. Isto é conseguido mediante a transparência semântica dos objetos para os computadores e humanos em tempo de execução e tempo de codificação. Especificamente, como é mostrado mais adiante, em XP.K a transparência semântica é atingida mediante o uso de UML para a apresentação de modelos não-ambiguos e significativamente corretos para computadores e humanos [18].

O desenvolvimento de SBC com XP.K se baseia portanto no paradigma de orientação a objetos no nível de implementação e na modelagem explícita de conhecimento no nível de projeto. Já que as ontologias são a ligação entre os *Agentes Especialistas* e o mundo do sistema, pode-se integrar a

tecnologia de Engenharia de Conhecimento em sistemas orientados a objetos e baseados em conhecimento.

Processo: XP.K é baseada em XP, que é um método ágil. Isso permite resolver os requisitos de processo apresentados anteriormente, já que estes métodos são baseados na comunicação e feedback rápido, assim como contar com uma comunidade de desenvolvedores com experiências valiosas e guías sobre como aplicar XP.
XP.K não descreve uma seqüência pré-definida de atividades. É definida por um conjunto de valores, princípios, e práticas que devem ser seguidas pelos desenvolvedores. A seguir, cada um desses elementos é explicado.

1. **Valores:** São os seguintes:

   Comunidade: Este valor sugere que as equipes devem ocupar recursos na construção e promoção da sua infraestrutura colaborativa. O valor da *Comunicação* em XP é requerido porque as ontologias são a ligação entre o domínio, as ferramentas e o sistema, além de comprometer os pontos de vista dos vários stakeholders com requisitos parcialmente contraditórios. O valor XP da *Humildade* significa que os desenvolvedores do sistema aceitam o seu conhecimento limitado e respeitam as atitudes, background, linguagem, e abordagens de outros desenvolvedores. Em particular, isto deve ser tomado em conta quando engenheiros e usuários especialistas trabalham juntos, já que os últimos geralmente não têm treinamento em técnicas de modelagem de conhecimento. O contrário pode provocar mal-entendidos e problemas sociais.

   Simplicidade: Os modelos de conhecimento criados no desenvolvimento de um SBC devem ser tão simples quanto possível, pois devem ser fáceis de entender e comunicar, em particular quando pessoas com diferentes formações como *Agentes Especialistas* (e.g. médicos) e programadores têm que trabalhar em conjunto.
   XP recomenda fazer pequenos investimentos no inicio, em vez de criar coisas complicadas que possam não ser utilizadas no futuro. No caso de modelos de conhecimento é recomendável manter modelos simples inicialmente para evitar o perfeccionismo incorreto e a ridigez prematura [39].

   Feedback: XP.K promove o feedback constante através de todo o processo de desenvolvi-

mento. Os Engenheiros de Conhecimento definem restrições semânticas sobre a ontologia que restringem o espaço de bases de conhecimento admissíveis. Estas restrições podem ser usadas para detectar inconsistências cedo no processo e guiar a aquisição de conhecimento. Os desenvolvedores escrevem testes unitários para todos os Métodos de resolução de problemas e outros módulos que possam falhar. Os protótipos são freqüentemente expostos a usuários potenciais e à administração do projeto para que se tenha noção do progresso e possa se ajustar a velocidade do projeto.

Coragem: A Coragem incentiva os desenvolvedores e modeladores de conhecimento a fazer mudanças relativamente grandes se necessário, de forma que se possa escapar de situações nas quais não se obtêm progressos mediante mudanças pequenas. Isto é provocado pela alta velocidade e pouco planejamento inicial que XP impõe ao desenvolvimento.

2. **Princípios:**

Como mostra a Figura 3.2, quase todos os princípios de XP são adotados por XP.K. Porem, alguns princípios são particularmente importantes para a modelagem de conhecimento.

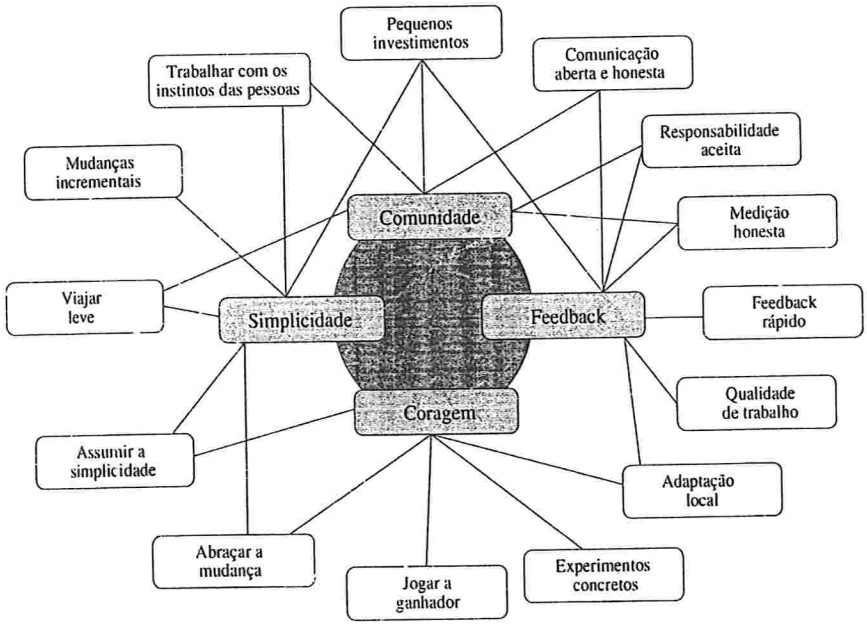

Figura 3.2: Os quatro valores de XP.K (centro) e os princípios que a suportam. (adaptado de [32])

Viajar leve Em vez de manter vários modelos com diferentes níveis de abstração, XP.K promove o uso de diferentes *visões*, em particular uma visão do domínio declarativo e uma visão do sistema executável (ver [33]).

Feedback Rápido e Experimentos Concretos As atividades de elicitação das ontologias e as suas restrições são apoiadas por testes automáticos de consistência. Estes testes são implementados em uma ferramenta de aquisição de conhecimento que pró-ativamente guia as atividades de modelagem do domínio.

3. **Práticas:**

XP.K se baseia nas práticas de XP (Figura 3.3) para as partes do sistema que não envolvem diretamente modelagem de conhecimento. Para o resto, especificamente modelagem da ontologia e aquisição de conhecimento, as seguintes práticas são aplicadas.

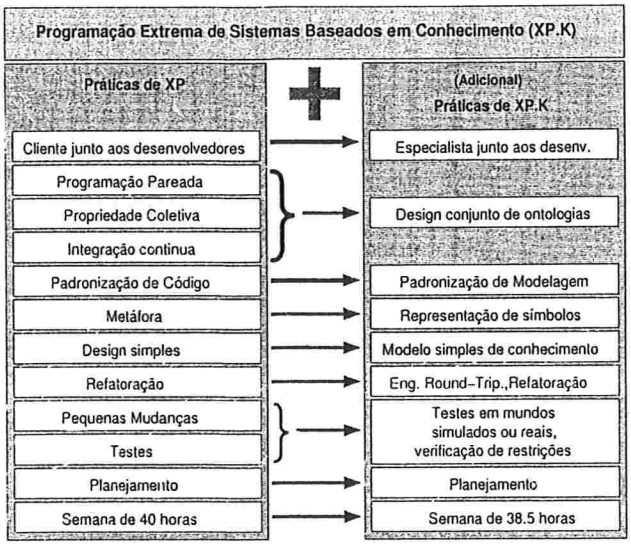

Figura 3.3: XP.K aplica práticas de XP para Sistemas baseado em Conhecimento. (adaptado de [32])

Usuário Especialista junto aos desenvolvedores: Similar à prática de *Cliente junto aos desenvolvedores* de XP, XP.K requer que pelo menos um *Agente Especialista* esteja disponível no local dos desenvolvedores. A tarefa deste usuário é prover informação acerca do domínio e o mundo dos Agentes Usuários. O tamanho das equipes em XP.K pode ser levemente maior que

em XP porque as equipes serão divididas em grupos de Agentes Especialistas, Engenheiros de Conhecimento, Desenvolvedores de ferramentas, e Desenvolvedores do sistema.

Design conjunto de Ontologias e Modelagem pareada: O Design conjunto de Ontologias tem contra-partidas em XP que são Programação Pareada, Propriedade coletiva de código e Integração Contínua. Na linguagem da ontologia orientada a objetos que é explicada mais adiante, os Agentes Especialistas usualmente não saberão como mudar as classes da ontologia, de maneira que a modelagem pareada é essencial.
Quanto à aquisição de instâncias da ontologia, a prática de Modelagem pareada sugere que os Agentes Especialistas modelem seu conhecimento em pares. Isto pode melhorar significativamente a qualidade do modelo, pois o conhecimento tácito e as preferênçias pessoais são generalizados, o que aproximará mais as perspectivas dos especialistas e dos engenheiros.

Engenharia Round-Trip: Esta prática é uma das tecnologias mais importantes em XP.K. Ela permite a geração de modelos de domínio de alto nível (UML) a partir de código fonte de baixo nível (Java) e vice-versa. A linguagem de modelagem XP.KL permite fazer a Engenharia Round-Trip porque consiste somente de conceitos que estão disponíveis em UML e em linguagens de programação como Java.

Padronização de Modelagem: Esta prática é a contra-partida da *Padronização de Código* em XP. Neste caso, os padrões de modelagem de conhecimento podem estabelecer formatos de nomes, guias, e regras sintáticas para a representação visual de elementos do modelo.

Representação de Símbolos: XP.K estende a noção de *Metáfora* em XP para representação comum de símbolos. *Representação de símbolos* (ou *Symbol Grounding* em inglês) [29] significa que a interpretação semântica de um sistema formal de símbolos (e.g. vocabulário do domínio) é embutida no sistema, por exemplo mediante representações "icônicas".
Estas representações de símbolos devem ser compartilhadas entre os desenvolvedores, para que os Agentes Especialistas conheçam as metáforas dos engenheiros e vice-versa, evitando conceitualizações erradas. Aliás, estes símbolos ajudam na comunicação entre os desenvolvedores através de um vocabulário bem definido e compreensível.

Testes e Verificação de Restrições: XP.K aplica a prática XP de escrever testes unitários para a modelagem de conhecimento. As ontologias e os Métodos de resolução de problemas são enriquecidas com métodos de teste que asseguram que a semântica projetada é implementada pelos modelos.

Modelo Simples de Conhecimento: Similar ao *Design simples* de XP, um modelo correto de conhecimento em XP.K é aquele que passa todos os testes sem violar as restrições, não tem lógica duplicada, tem a menor quantidade de conceitos, propriedades, e instâncias, e provê transparência semântica para os desenvolvedores e computadores. Os modelos de conhecimento devem representar somente o necessário para resolver as tarefas dadas.

Padrões de Projeto e Refatoração: A prática XP de *Refatoração* pode ser aplicada também à modelagem de conhecimento. No nível de classes ontológicas, a refatoração significa aplicar os padrões de refatoração da programação orientada a objetos. No nível de instâncias, a refatoração significa combinar elementos duplicados do modelo, para melhorar os modelos visuais, ou para estender a documentação dos elementos mais estáveis.
Outra aplicação da refatoração de modelos de conhecimento é a generalização de modelos existentes para sua futura reutilização. Também podem ser aplicados *Padrões de Projeto* para modelar o conhecimento. Por exemplo, o padrão *Composite* [26] é aplicável a muitas situações onde os conceitos formam uma hierarquia.

Planejamento: Esta prática sugere a utilização de histórias para guiar o processo de desenvolvimento, o que equilibra as prioridades dos usuários com as tecnicamente possíveis. Em XP.K, os Agentes Especialistas empilham estas historias de acordo com as suas prioridades e os desenvolvedores acrescentam estimativas sobre os recursos requeridos. Isto é repetido até atingir um equilíbrio. Assim, os Agentes Especialistas decidem conjuntamente o cronograma do projeto.

Linguagens:

XP.KL é uma linguagem simples para a declaração de ontologias e bases de conhecimento. A principal meta de XP.KL é integrar os valores, princípios, e práticas de XP.K. Em particular, XP.KL

apóia a *Simplicidade*, *Comunidade*, *Feedback rápido* e *Viajar leve*. A Figura 3.4 mostra as classes de XP.KL em notação UML/MOF.

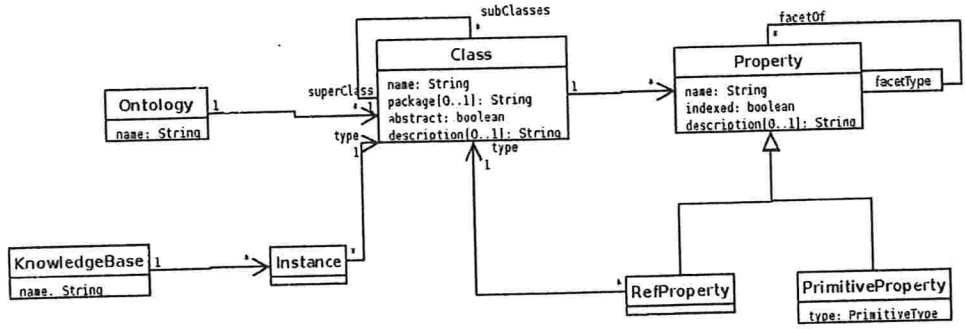

Figura 3.4: O metamodelo da linguagem de modelamento de conhecimento XP.KL [32]

XP.KL requer que as plataformas de implementação (e.g. Java) dêem suporte a *Reflexão*. A Reflexão permite acessar meta-dados sobre objetos, classes e propriedades em tempo de execução. Usando estes meta-dados, os programas podem encontrar dinamicamente os atributos das bases de conhecimento e classes, para que algoritmos genéricos e ferramentas possam ser construídas. A Reflexão é uma técnica essencial para a implementação da transparência semântica.

Apesar de XP.KL ser menos expressivo que linguagens especializadas de representação de conhecimento, a prática industrial com software orientado a objetos tem mostrado que as suas primitivas de modelagem são apropriadas para modelar diversos domínios.

Ferramentas:

Editores de Ontologias (Classes) Devido ao meta-modelo que XP.KL propõe, as classes da ontologia podem ser visualizadas e editadas com editores UML. Uma das ferramentas mais modernas de Engenharia Round-Trip para UML é Together for Java [11].
As facilidades de edição visual destas ferramentas UML simplificam a comunicação com os Agentes Especialistas. Porém, as ferramentas CASE têm um alto número de características e oportunidades para cometer erros, portanto os Agentes Especialistas não deverão utilizá-las por conta própria. Já que XP.K implementa a prática de Design Conjunto de Ontologias, não há problemas em deixar um Engenheiro de Conhecimento fazer as mudanças.

Editores de Bases de Conhecimento (Instâncias) O meta-modelo de conhecimento de XP.K tem suporte explícito para Reflexão. Esta propriedade é usada para o desenvolvimento de vários editores *genéricos* de instâncias, que podem operar em qualquer estrutura de dados em Java que siga um conjunto de regras de formatação (a especificação JavaBeans [37], que implementa XP.KL). Estes editores são parte da plataforma KBeansShell [33] para o desenvolvimento de ferramentas de aquisição de conhecimento.
KBeansShell é extremamente flexível e aberto. A customização do processo de modelagem de conhecimento por meio de componentes definidos pelo usuário é provavelmente uma das características mais notáveis de toda a abordagem Round-Trip em XP.K, porque as classes Java da ontologia podem ser estendidas com código extra sem quebrar a sua transparência semântica.

3.3 RapidOWL

RapidOWL é uma metodologia que se baseia em XP.K [7]. A principal diferença é o uso de RDF como linguagem de conhecimento com o intuito de usar sentenças tão pequenas quanto possível de forma a construir a base de conhecimento incrementalmente.

A Engenharia de Conhecimento se vê afetada quando usada no âmbito da Web Semântica. Estas mudanças afetam o projeto de RapidOWL e são descritas a seguir:

1. **A Engenharia de Conhecimento não é um negócio em si mesma.** A Engenharia de

Conhecimento é geralmente implementada quando o conhecimento utilizado para criar algum bem ou serviço está distribuído, devido aos usuários e comunidades estarem localizados em pontos distantes. Este tipo de cooperação é comum em Organizações Virtuais [3].

2. **Falta de uma Única Serialização de Conhecimento.** Os métodos ágeis fazem bastante de uso de sistemas de versões (e.g. Subversion, CVS). Estes são adequados para construir sistemas de software, pois a ordem das linhas de código em um programa são importantes, não sendo este o caso em bases de conhecimento onde a ordem dos axiomas é irrelevante.

3. **Separação parcial das partes.** Muitos métodos ágeis se baseiam na presença do cliente no local de desenvolvimento. Mas quando as partes se encontram separadas espacialmente, o uso da Engenharia de Conhecimento baseada em ferramentas se torna importante.

4. **Envolvimento de um número grande de membros.** Os cenários da Engenharia de Conhecimento diferem dos cenários de desenvolvimento de software: é usualmente crucial integrar um grande número de especialistas, engenheiros de conhecimento e finalmente os usuários das bases de conhecimento. Tecnologias Web e a Internet podem apoiar esta integração servindo para que um método ágil de engenharia de conhecimento possa integrar estes especialistas no processo de engenharia.

Paradigmas Além de ser ágil, este método se baseia nos padrões de representação da Web Semântica. Sendo assim, ela deve refletir a natureza interconectada e distribuída da Web e reconhecer as sentenças RDF como os menores blocos de construção de bases de conhecimento na web Semântica. Assim, o paradigma de representação de conhecimento para ontologias é substituído por representações de conhecimento baseadas no *modelo de dados da Web Semântica.*

Processo As metodologias ágeis possuem um conjunto de valores nos quais se baseiam os princípios que definem o processo de engenharia, assim como as práticas para estabelecer estes princípios na vida diária.
Vamos descrever estes elementos.

Valores RapidOWL adota os valores de XP, especificamente a *Comunicação* (desenvolvimento conjunto de ontologias), *Feedback* (promover a evolução), *Simplicidade* (incrementar a manutenção da base de conhecimento) e *Coragem* (ser capaz de escapar de caminhos sem saída). Porém, RapidOWL combina os valores de Feedback e Comunicação no valor de *Comunidade*, que inclui as construções sociais que dão suporte à comunicação e ao feedback. Além dos valores

XP, RapidOWL inclui o valor da *Transparência*, que garante a redução do custo de mudança da base de conhecimento. Isto porque para cumprir o objetivo de RapidOWL de construir bases de conhecimento de forma incremental e distribuída, ela precisa de práticas e técnicas para reduzir o custo dessas mudanças.

Princípios Os princípios estão parcialmente baseados nos critérios de design do Wiki [19]. Estes princípios incluem *suposição do mundo aberto*, promoção de *mudanças incrementais*, métodos de *criação uniforme* para ambos o modelamento e a aquisição de conhecimento, *desenvolvimento observável* e *feedback rápido* (ver [4] para mais detalhes).

Práticas As práticas são baseadas nas práticas de XP.K e promovem o suporte aos princípios apresentados. Devido às características específicas de Engenharia de Conhecimento apresentadas anteriormente, nem todas essas práticas serão adotadas: *Design Conjunto de Ontologias*, *Integração de Informação* (basear a elicitação de conhecimento em informação existente), *Geração de visões* (prover visões específicas do domínio para usuários humanos e sistemas de software) e *Evolução de Ontologias* (facilitar a adoção de modelamentos e a migração das instâncias correspondentes).
RapidOWL transforma os Agentes Especialistas em Engenheiros de Conhecimento de tempo parcial, mantendo as práticas tão simples quanto possível e propondo estratégias para suporte mediante ferramentas.

Ferramentas Para dar suporte aos princípios de RapidOWL foram desenvolvidas duas ferramentas:

pOWL Esta é uma plataforma para desenvolvimento de aplicações para a Web Semântica [5]. Ela provê livrarias para a criação e manutenção colaborativa de ontologias em OWL, alem de conter componentes graficos ou "widgets" para a construção de aplicações. Possui também um sistema de controle de versões para sentenças RDF [6].

OntoWiki Esta é uma ferramenta construida sobre pOWL para dar suporte a cenários ágeis e distribuidos de engenharia de conhecimento [5]. OntoWiki facilita a presentação visual de uma base de conhecimento como um mapa de informação, com diferentes vistas das instâncias.

Capítulo 4

Método Ágil de Desenvolvimento de AC

Neste capítulo propomos um método ágil de engenharia de conhecimento baseado em XP.K para desenvolver Sistemas Baseados em Artefato de Conhecimento (SBAC) como sugerido em [17].

Vamos retomar a noção de AC dada no Capítulo 2. Para o autor A poder construir um artefato R feito de conhecimento (Figura 2.1), deve existir um "método" para elicitar o conhecimento relevante correspondente.

A intenção do método aqui descrito é tornar computacional o AC detectado. O SBAC é portanto um AC embutido na forma de um sistema de software. O nosso paradigma central é o aproveitamento das características de XP.K e extendé-lo para lidar com o aspecto emergente dos AC. O nosso método recebe também influência de RapidOWL no uso de linguagens da Web Semântica para definir e compartilhar os modelos baseados em AC.

Os valores, princípios e práticas do método proposto são apresentados nas próximas seções. Esta proposta é baseada na nossa experiência prévia descrita no Capítulo 2 e foi publicada em [25].

4.1 Valores

Comunidade Os Engenheiros de Conhecimento e Desenvolvedores do SBAC precisam enxergar as suas atividades de duas formas: (1) como construtores de uma Comunidade de Prática (CDP) [1] que desenvolve um SBAC, e (2) como cientes da existência de uma outra *CDP de Agentes*. No primeiro caso é necessário que se mantenha uma comunicação fluída, como sugere o valor XP.K da *Comunicação*, pois isso garante a estabilidade da Comunidade. Destaca-se o papel do

[1] Neste capítulo vamos usar o termo "Comunidade" para nos referir a "Comunidade de Prática" ou "CDP"

AC como principal meio de comunicação entre os membros da equipe. No segundo caso eles devem enxergar a organização como um conjunto de Comunidades para ter uma melhor noção de quem são as pessoas envolvidas no desenvolvimento do SBAC.
O princípio da *Participação periférica* fala do tipo de interação que deve existir entre a Comunidade dos Engenheiros de Conhecimento e Desenvolvedores e a *Comunidade de Agentes*. A prática do *Design Conjunto do AC* detalha as atividades envolvidas nessa interação.
É importante ter em conta o valor da *Humildade* e o *Respeito* em XP, pois traz para aos *Engenheiros de Conhecimento* e *Desenvolvedores* um código de conduta para com os membros da *Comunidade de Agentes*, que geralmente não têm treinamento em programação ou modelagem de conhecimento.

Simplicidade A simplicidade é uma conseqüência do uso de um AC no desenvolvimento de um SBAC. De fato, já que um AC é um padrão cognitivo possuido pela Comunidade, o uso da gramatica derivada diretamente do AC promove uma integraçao simples entre os membros da Comunidade.

Uma recomendação importante do XP.K sugere fazer o projeto dos modelos de conhecimento o mais simples possível. Estamos considerando esta sugestão no nosso método porque apesar que a interaçao é simples por causa do AC, a fase de design (definida por Winograd em [54]) é díficil e complexa. O design é ainda mais complexo porque o desenvolvimento do SBAC requer todas as três dimensões do modelo de conhecimento baseado em AC (intensional, extensional e de raciocínio, veja 2).

Feedback Uma ferramenta que dê o suporte à fase de Design do AC é a principal via para os envolvidos no desenvolvimento do SBAC obter um feedback. Esta ferramenta deve permitir configurar os testes para todas as dimensões do modelo baseado em AC, assim como uma interface simples e intuitiva que fornecerá informação sobre o estado do AC.
Além disso, esta ferramenta deve gerar código fonte em alguma linguagem de programação (e.g. Java, C#) refletindo a arquitetura de um SBAC pronta para ser integrada com alguma GUI.

Coragem A Coragem é necessária para abandonar o design de um AC quando este não satisfaz os requisitos da *Comunidade de Agentes* onde se pretende desenvolvé-lo.
Cabe ressaltar que apesar de muito código fonte ser abandonado, os resultados da análise ontológica e epistemológica do domínio permanecem intactos, pois estes são armazenados de forma independente da instância do AC, podendo ser reutilizados em outro AC.

4.2 Princípios

Muitos dos princípios de XP.K são reutilizados aqui. Porém, alguns deles afetam mais diretamente o desenvolvimento de um AC e são analisados com maior profundidade. A Figura 4.1 mostra os valores e princípios do nosso método.

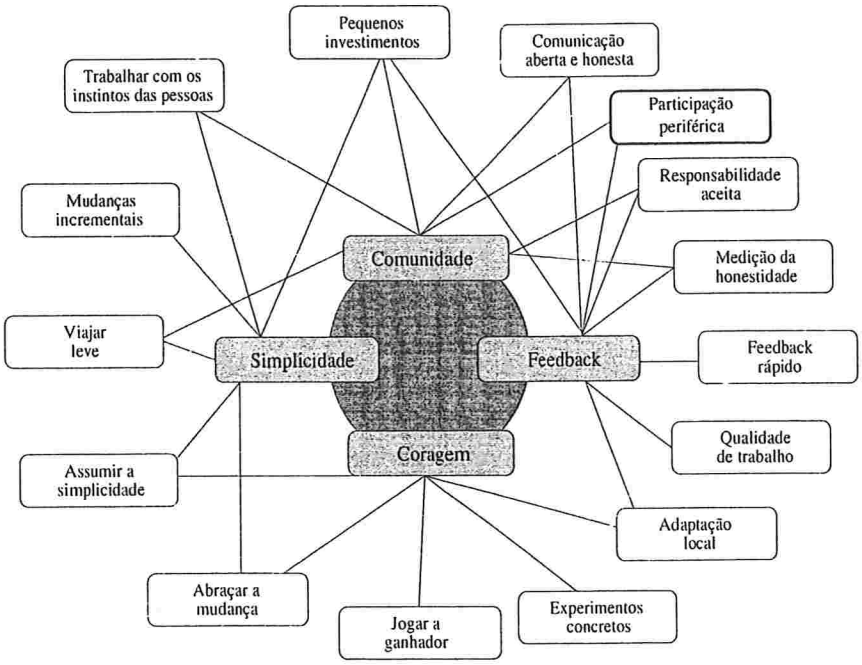

Figura 4.1: Os Valores propostos e os principios que dão suporte ao nosso método. Alguns destes valores e princípios foram redefinidos do ponto de vista de uma Comunidade e AC (adaptado de [32])

Trabalhar com os instintos das pessoas Já que o nosso paradigma está baseado na teoria de CDP, o conhecimento tácito dos membros de uma Comunidade é importante para fazer o Design de um AC.

Feedback rápido e Experimentos concretos Como resultado do uso de uma ferramenta como a sugerida no valor do *Feedback*, os experimentos com código fonte são mais rápidos, já que a ferramenta pode gerar código fonte baseado no framework arquitetural de SBAC (ver capítulo 2) e no modelo de conhecimento baseado em AC.

Esta ferramenta deve realizar também testes automáticos e verificação de consistência lógica no modelo baseado em AC para prevenir modificações custosas de última hora.

Abraçar a mudança É possível que muito do código fonte produzido (ou gerado pela ferramenta) seja abandonado para recomeçar o design de um novo AC. Este principio dá o suporte para o valor da Coragem.

Participação periférica O método que estamos propondo considera duas Comunidades principais. As pessoas pertencendo à primeira sao os Usuarios e Especialistas, enquanto os Desenvolvedores e Engenheiros de Conhecimento conformam a segunda.

A interação entre os membros de ambas comunidades começa na periferia. No primeiro cenário, a Comunidade de Usuários e Especialistas aceita um Engenheiro de Conhecimento ou Desenvolvedor como um novo usuario da comunidade. Ele pode chegar até o núcleo da Comunidade onde o conhecimento é detido pelos *Especialistas*. Esta atividade ajuda os *Engenheiros e Desenvolvedores* a ter uma melhor noção do âmbito de desenvolvimento do SBAC.

As práticas de *Especialista junto aos desenvolvedores* e *Design conjunto do AC* são o caso contrario onde o *Especialista* faz parte da *CDP de Engenheiros e Desenvolvedores*. Neste caso a interação é também periférica, pois os Especialistas aprendem gradualmente técnicas de programação e modelagem de conhecimento ao mesmo tempo que os modelos de conhecimento são criados.

Viajar leve O nosso método promove a manipulação de um AC através de uma ferramenta com uma Interface de Usuário apropriada. Esta interface apresenta de forma efetiva as dimensões do modelo baseado em AC. O AC é o principal objeto a ser manipulado durante o desenvolvimento do SBAC. Podem existir outros artefatos como diagramas UML ou planilhas eletronicas sempre que o AC seja mantido como o principal modelo.

4.3 Práticas

Conforme foi adotado em XP.K, o nosso método utiliza as práticas XP para as partes do SBAC que não envolvem conhecimento. São adaptadas algumas das práticas XP.K para o design de modelos baseados em ACs (Figura 4.2).

Testes num mundo simulado ou real, verificação de restrições Promovemos o uso de linguagens da Web Semântica para expressar as dimensões que compoem um modelo baseado em AC

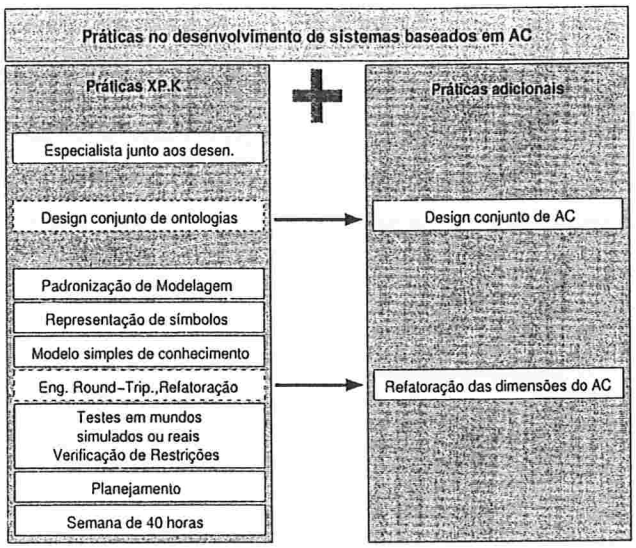

Figura 4.2: As práticas que conformam o nosso método estão inspiradas nas práticas de XP.K. Algumas delas foram substituídas como é o caso de *Design conjunto de Ontologias*. (adaptado de [32])

porque estes são amplamente adotados em aplicações na indústria e em atividades de pesquisa. Facilita também a difusão de modelos baseados em AC na Internet podendo ser usados por Comunidades virtuais. Motores de inferência como Flora [57] ou RACER [28] ou KAON2 [12] permitem fazer a verificação da consistência de uma ontologia em OWL-DL.

Além de testar a ontologia em OWL-DL, as regras epistemológicas expressas em SWRL [38] podem também passar por testes (a combinação de OWL-DL e Regras SWRL DL-Safe esta implementada em Pellet).

Finalmente o Método de resolução de problemas pode ser expresso na linguagem OWL-S [23]. Esta linguagem permite descrever serviços web semânticos desde o nível mais alto (e.g. Inputs) até o mais baixo (execuçao de serviços web). Nesta descrição um conjunto de expressões (e.g. If-Then-Else) são usados para definir o fluxo de execução do serviço web semântico. Este fluxo de execução pode ser reutilizado para definir a sequência de passos em um Método de resolução de problemas. Mais ainda, motores para OWL-S ([43], [27]) podem executar estas descrições.

Representação de Símbolos O AC pode ser visto como uma metáfora para a equipe de desenvolvimento. A metáfora é conduzida pela mensagem que o AC traz (e.g. no caso do *Maintenance and*

Repair, esta mensagem é dada parcialmente pelos elementos gramaticais "⟨device⟩","⟨part⟩" e "⟨damage⟩").

Modelo Simples de AC Um modelo baseado em AC inclui diferentes classes de modelos de conhecimento (e.g. ontológico) que devem ser modelados tendo em mente as necessidades atuais. A prática de *Rapid Feedback* provê informação acerca da consistência destes modelos para guiar a modelagem.

Design Conjunto de AC No XP.K, a prática de Design conjunto de Ontologias foi projetada para que o engenheiro e o especialista pudessem encontrar uma representação razoável da ontologia do domínio. No nosso caso o objetivo é fazer o design de um AC.

A meta é construir o modelo baseado em AC composto pela gramática, sentenças e Método de resolução de problemas, que expressam de maneira formal o conhecimento ontológico e epistemológico da Comunidade assim como um método para processar estes elementos.

A gramática é projetada depois de uma análise ontológica e epistemológica do domínio da aplicação. É também derivada a partir de um vocabulário informal dependente do domínio (i.e. a linguagem profissional) usada pelos *Agentes Especialistas.*

Em outras palavras, o modelo baseado em AC é um modelo de conhecimento especificado mediante sentenças escritas usando uma linguagem projetada.

Entidades, propriedades, relações e processos são definidos pela gramática da linguagem formando a dimensão intensional do modelo.

Para evitar a ambigüidade do modelo devido ao uso de linguagem natural usada pela *CDP*, é obrigatório que essa linguagem seja formal. O *Engenheiro de Conhecimento* promove a participação dos *Agentes Especialistas* nas duas tarefas: as atividades que levam à definição da linguagem e também a especificação do modelo baseado em AC inicial que será implementado no SBAC e mantido pelo *Especialista Mantenedor.*

Padronização de Modelagem É definida pelo *Engenheiro de Conhecimento* em conjunto com o *Agente Especialista.* Depende muito da linguagem técnica usada pela *CDP de Agentes* e terá efeito sobre a definição do modelo baseado em AC.

Refatoração das dimensões do AC Em XP.K foi definida a prática de *Refatoração e Engenharia Round-Trip*, que inclui técnicas para gerar código fonte a partir de modelos em UML e viceversa.

Isto nao pode ser aplicado no nosso caso porque estamos usando OWL-DL como linguagem de metamodelo. No nosso caso a refatoração poderá acontecer em cada dimensão do modelo baseado em AC e ser de dois tipos: (1) Externa: cada dimensão é uma entidade independente em OWL-DL (e.g. ontologia) (ela pode ser reutilizada no design de outro AC), (2) Interna: podem se utilizar *Padrões de Conhecimento* [15] para aproveitar alguns padrões recorrentes na modelagem de ontologias.

4.4 Aspectos na Modelagem de ACs

No Capítulo 2 e em [25] foram descritos dois AC concretos relacionados a dois projetos diferentes. Tanto *Design by Adaptation* como o *Maintenance and Repair* têm nos guiado no desenvolvimento de sistemas de software específicos, e estas experiencias formam a base da nossa proposta para o metodo exposto neste capítulo.

A teoria de AC dada no Capítulo 2 pode ser ainda mais enriquecida. Por exemplo, a relação entre AC concretos e abstratos (como o *Maintenance and Repair*) pode ser mais profundamente analisada e explicada: O AC abstrato *Maintenance and Repair* tem como propósito P diagnosticar e reparar dispositivos complexos. A sua implementação R pode ser representada mediante elementos ontológicos presentes na gramática e no método de resolução de problemas, descrevendo como as sentenças escritas usando esta gramatica devem ser consideradas. No domínio dos carros, este AC é reificado com conhecimento específico (e.g. reparo de carros) para cumprir com o propósito do conhecimento específico de reparar um carro. Isto resulta em um AC concreto com um propósito P diferente (consertar danos no motor de um carro) e uma implementação I (o sistema de software ou SBAC).A Figura 4.3 mostra esta relação.

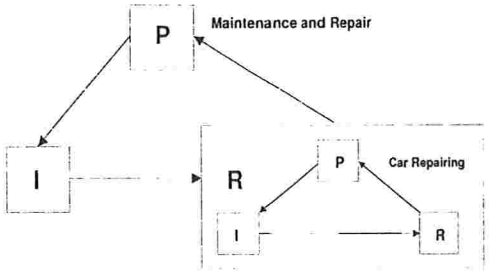

Figura 4.3: O AC Maintenance and Repair reificado no dominio de reparo de carros.

Existe também uma relação menos trivial entre todos os AC abstratos e o "AC de AC" que chamamos de "metaAC". A Figura 4.4 mostra este "metaAC". O seu propósito P é criar um AC e a sua implementação R é um AC mais concreto (e.g. *Design by Adaptation, Maintenance and Repair*). Poderíamos continuar subindo nos niveis de abstração, até chegar em um domínio geral como Design ou Diagnóstico.

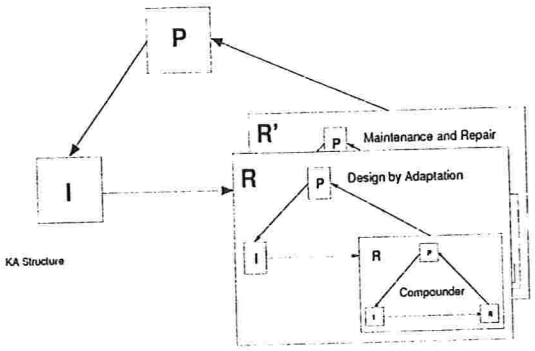

Figura 4.4: O metaKA e sua relação com o conjunto de AC

No primeiro nível de análise mostrado na Figura 4.3, implementamos uma ferramenta (Capítulo 5) para ajudar no desenvolvimento de SBAC. Acreditamos que a utilização de um AC junto com um método ágil para desenvolver tal tipo de sistemas permite aos desenvolvedores construi-los de forma mais efetiva e rápida. Um dos principais problemas no desenvolvimento de sistemas baseados em conhecimento é a falta de ferramenta para testar o conhecimento sendo elicitado. No caso de um modelo baseado em AC este torna-se um desafio ainda maior já que diferentes modelos correspondendo a diferentes sistemas lógicos (e.g. SWRL e OWL-DL) devem ser testados. Algumas ferramentas existentes podem realizar testes parciais (e.g. Protègè e SWOOP).

Além disso, temos explorado a relação entre Design e IA [55] para poder produzir uma GUI simples que dirija os desenvolvedores na elicitaçao de conhecimento de propósito especial, mais especificamente, na recuperação das descrições gramaticais e do Método de Resolução de Problemas a partir do AC codificado e na tradução destes na interface de usuário da ferramenta.

Capítulo 5

Modelo e Implementação da Ferramenta

Neste capítulo mostramos a implementação de uma ferramenta para apoiar o método ágil apresentado no capítulo anterior. Especificamente, esta ferramenta tenta ajudar na elicitação de modelos baseados em AC usando BNF como linguagem de representação e OWL como formato de armazenamento para estes modelos.
A ferramenta está implementada em um plugin para o Protègè 4.0 versão Alfa. Esta versão de Protègè usa o framework OSGi para a execução e desenvolvimento dos seus plugins.
Na seção 5.1 mostramos um conjunto de ontologias em OWL que servem como metamodelos para a nossa ferramenta poder elicitar os AC. Devido à hierarquia que existe entre os modelos de AC (Figura 4.4), estas ontologias fogem da semântica dada por OWL-DL mas tentamos manter estes modelos o mais simples possível para permitir a motores de inferência (e.g. Pellet) fazerem consultas.
Logo na seção 5.2 mostramos a arquitetura interna do nosso plugin. Foi implementada uma arquitetura MVC onde o "Model" contém classes que refletem o modelo ontológico de AC. Os componentes "View" e "Controller" são implementados usando extensões dadas pelo framework Protègè.

5.1 Modelo de Artefatos de Conhecimento

Como foi sugerido pelo método proposto no capítulo 4 e mostrado no AC "Maintenance and Repair", as gramáticas BNF são um formalismo fácil de aprender pelos usuários e de ser implementado em modelos de bases de dados ([17]). Ao mesmo tempo, o nosso método propõe o uso linguagens da Web Semântica para formalizar modelos baseados em AC. Devemos levar em conta que os atuais motores de inferência que suportam OWL somente computam sobre o sub-conjunto OWL-DL (e.g. Pellet).

Com essas restrições propomos uma ontologia em OWL para os modelos baseados em AC[1]. Esta ontologia tenta abstrair as dimensões do modelo baseado em AC segundo definido no capítulo 2 (ver [17] para mais detalhes). A Figura 5.1 mostra esta ontologia:

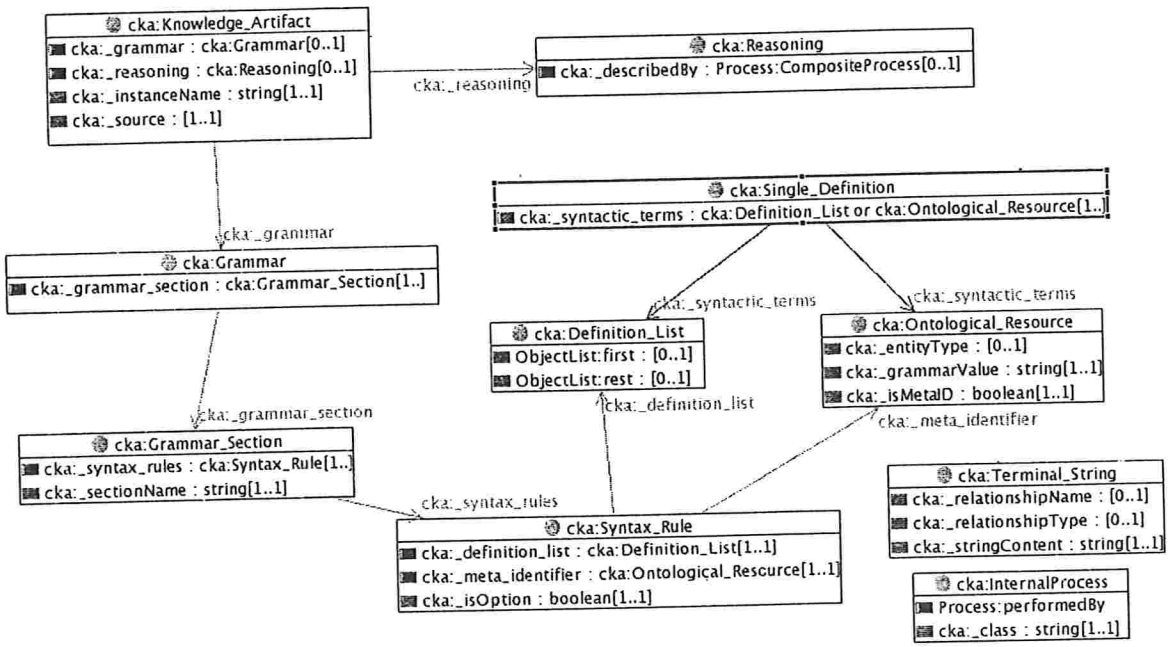

Figura 5.1: Ontologia de modelos baseados em AC

O principal conceito é a classe *Knowledge Artifact* que representa o modelo baseado em AC. Esta classe contem por sua vez as dimensões de raciocínio e gramatical que estão representadas pelas classes *Reasoning* e *Grammar* respetivamente.

A intenção da classe *Reasoning* é de apontar para uma instância da classe *CompositeProcess* em OWL-S que sirva para especificar o método de resolução de problemas que da suporte computacional ao artefato.

A gramática representada pela classe *Grammar* está organizada em seções gramaticais representadas pela classe *GrammarSection*, estas por sua vez contém regras sintáticas. Estas regras são instancias da classe "SyntaxRule" que contem três propriedades:

[1]Disponível em http://www.ime.usp.br/~gsalazar/cka.owl

Meta Identificador Esta propriedade define o nome da regra que sera utilizada na etapa de elicitação (isto será esclarecido quando a utilização da ferramenta for mostrada no seguinte capítulo). O contra-domínio desta propriedade é a classe *OntologicalResource*.

Lista de definições Esta propriedade define a estrutura da regra sintática que é composta por metaidentificadores (classe *OntologicalResource*), definições simples (classe *SingleDefinition*), ou strings terminais (classe *TerminalString*) ou outras listas de definições. O rango desta propriedade é a classe *DefinitionList* [2].

Lista opcional Esta propriedade booleana especifica se a regra contem ou não uma lista de listas de definições.

A Figura 5.2 mostra a construção de uma regra sintática usando todos os elementos descritos anteriormente.

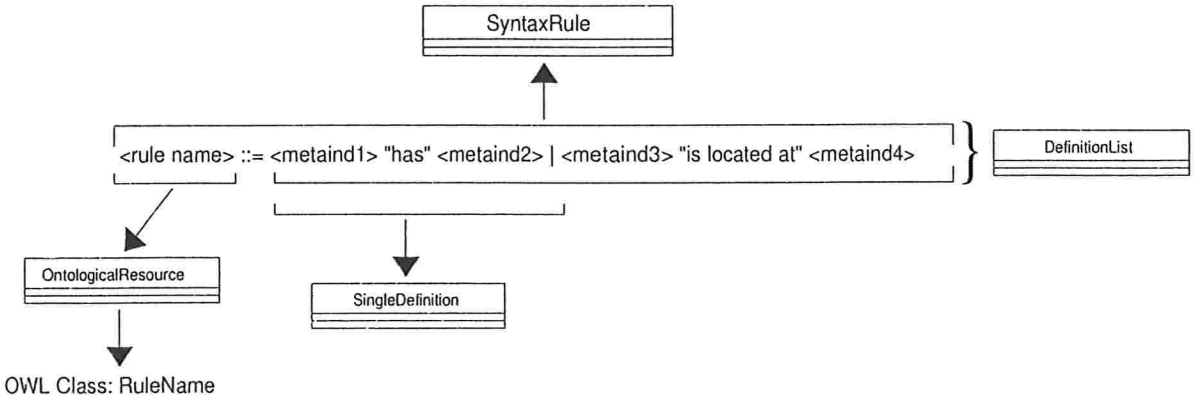

Figura 5.2: Exemplo de construção de regra sintática

As classes que compõem a lista de definições contem cada uma uma semântica distinta:

OntologicalResource Esta classe define meta identificadores para regras sintáticas que podem ser associadas a alguma classe OWL.

[2] Esta é uma subclasse de *List* definida pela ontologia http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl. Esta classe define uma lista com propriedades "first" e "rest" com rangos não definidos

TerminalString Esta classe permite representar arranjos de caracteres que ligam dois meta-identificadores e permitem criar a regra sintática. Alem disso elas podem ser associadas a propriedades OWL.

SingleDefinition Esta classe serve como contenedor para outros elementos gramaticais como listas de definições ou meta identificadores.

Na Figura 5.2 a instância de "SingleDefinition" ressaltada contem por sua vez mais elementos gramaticais. Estes elementos são mostrados com maior detalhe na Figura 5.3:

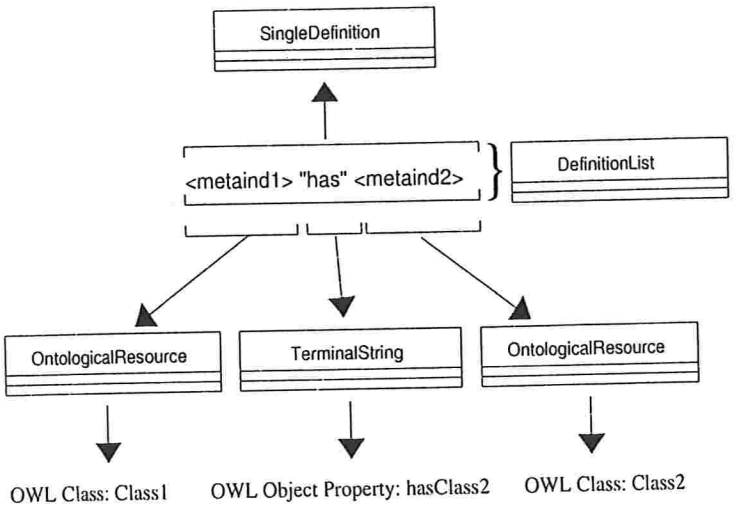

Figura 5.3: Exemplo de construção de regra sintática (continuação da Figura 5.2)

A semântica da regra sintática permite criar sentenças com base nas instâncias das classes associadas com os meta-identificadores e as propriedades relacionadas com os identificadores terminais. Estas sentenças constituem a dimensão de sentenças do modelo baseado em AC.

5.2 Implementação e Arquitetura do Plugin

O nossa ferramenta é implementada em um plugin para o Protègè 4.0 versão Alfa[3] e implementa o padrão arquitetural Model-View-Controller (MVC). Esta versão de Protègè foi implementada usando a especificação OSGi [1] e usa a plataforma Eclipse Equinox[4] que é uma implementação em Java do

[3]Disponível em http://protege.stanford.edu/download/prerelease-alpha/index.html
[4]http://www.eclipse.org/equinox

OSGi.
O Protègè esta composto por um conjunto de plugins (ou bundles) que definem pontos de extensão para o desenvolvedor criar o seu próprio plugin. Estes plugins são descritos por nomes de pacotes. A seguir listamos os plugins mais importantes usados neste projeto:

1. **org.protege.common**: Que define propriedades gerais para a execução do Protègè como a pasta onde esta instalado.
2. **org.protege.editor.core.application**: É o plugin mais importante porque define toda a arquitetura e os pontos de extensão do Protègè .
3. **org.protege.editor.owl**: Implementa as interfaces mais especificas para criar um editor para OWL. Faz uso de varios pontos de extensão do plugin anterior.
4. **org.semanticweb.owl.owlapi**: Importa o OWL API[5] que é uma API para manipulação eficiente de ontologias.
5. **com.owldl.pellet**: Importa o Pellet [44] que é um motor de inferência para OWL-DL.

Todos estes pacotes foram usados na implementação do nosso plugin, especialmente o segundo porque foram utilizados pontos de extensão deste plugin para definir elementos visuais.

Arquitetura

Podemos dividir a arquitetura do nosso plugin em dois aspectos:

Externo que define os pontos de extensão que conectam o plugin com o Protègè .Estes pontos de extensão nos permitem definir os elementos visuais da ferramenta que conformam o "View" e o "Controller" do padrão arquitetural MVC.

Interno que define o funcionamento interno. Neste componente são implementadas as classes da camada "Model" na arquitetura MVC.

Aspecto Externo

No aspecto externo foram usados pontos de extensão do plugin **org.protege.editor.core.application** para definir elementos visuais. As classes que implementam estes elementos são:

[5]Disponível em http://owlapi.sourceforge.net

KAGrammarTab Esta classe usa o ponto de extensão "org.protege.editor.core.application.WorkspaceTab" e cria uma aba para conter todas as vistas descritas abaixo.

KASectionsAndRulesView Esta vista, assim como as outras, implementa o ponto de extensão "org.protege.editor.core.application.ViewComponent" e mostra as seções gramaticais e regras sintáticas presentes no modelo.

KARuleDescriptionView Mostra a descrição da regra sintática, isto é as listas de definições.

KASentencesView Mostra as sentenças produto da especificação da regra sintática.

ConvertToKAMenuAction Este elemento implementa uma ação que converte a ontologia ativa em um AC mediante a importação da ontologia de um AC concreto (e.g. "Maintenance and Repair") como será explicado no seguinte capítulo.

A Figura 5.4 mostra a disposição destes componentes visuais.

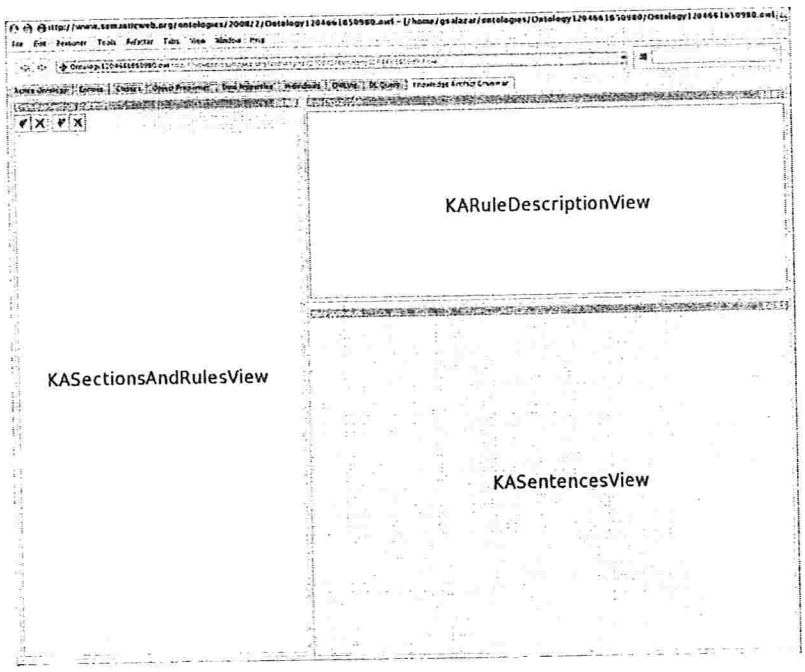

Figura 5.4: Esquema visual do plugin

Apesar destas classes usarem os pontos de extensão explicados anteriormente, elas se baseiam em classes do plugin **org.protege.editor.owl** para reutilizar funcionalidade já testada (e.g. seleção de um individuo OWL). A Figura 5.5 mostra estas dependências assim como outras classes importantes que provêm a funcionalidade da interface de usuário. Vale a pena mencionar que a classe **KASectionsAndRulesView** implementa a interface **KAViewModel** e tem o seu comportamento definido pela classe **AbstractOWLIndividualViewComponent** [6] e pode ser associada a instâncias de algumas classes do pacote **edu.usp.liamf.ka.model**. A interface **KAViewModel** força à classe que a implementa a fazer o papel de "Observer" escutando quaisquer mudanças no modelo associado (e.g. instâncias da classe **SyntaxRule**). No caso específico de **KASectionsAndRulesView**, ela escuta mudanças da gramática (e.g. criar novas seções gramaticais), das seções gramaticais e regras sintáticas quando estas são criadas ou removidas do modelo. Estas mudanças são capturadas e transformadas em axiomas OWL mediante a classe **KA2OWLProcessor** que implementa o padrão "Visitor". Esta classe aplica os cambios na ontologia ativa em OWL dependendo do tipo de objeto mudado no AC (e.g. instância de **GrammarSection**) e delega estas mudanças ao framework do Protègè .

[6] Este tipo de implementação de uma classe onde a interface e o comportamento podem variar corresponde com o padrão de projeto "Bridge". Este padrão tem sido usado ao longo de todo o plugin.

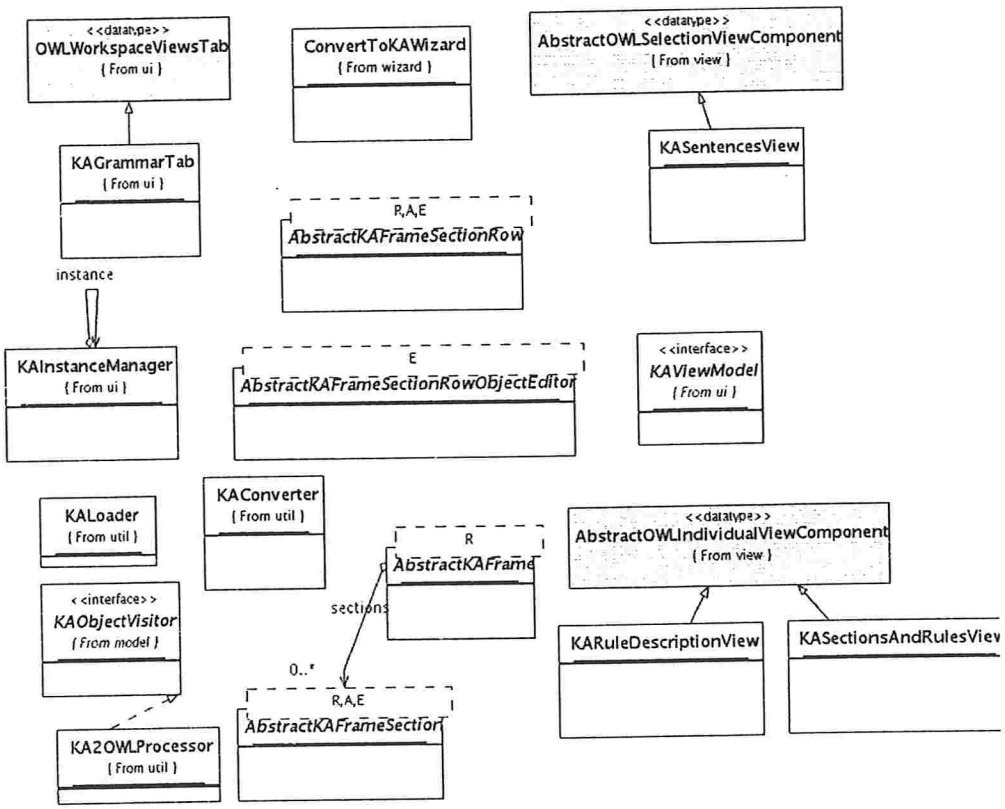

Figura 5.5: Principais classes do pacote **edu.usp.liamf.ka.ui**

As classes **KARuleDescriptionView** e **KASentencesView** usam listas para apresentar o conteúdo da regra selecionada na classe **KASectionsAndRulesView**. Estas listas estão organizadas hierarquicamente como mostra o esquema da Figura 5.6. Na implementação, os elementos "Frame", "Section", "Row" e "RowEditor" corresponde com as classes abstratas **AbstractKAFrame**, **AbstractKAFrameSection**, **AbstractKAFrameSectionRow** e **AbstractKAFrameSectionRowEditor** respetivamente.

Dependendo de onde sejam usadas estas classes é necessário derivar uma nova classe como é o caso de **KARuleDescriptionView** onde o "frame" só tem uma seção que mostra listas de definições da regra sintática atualmente selecionada.

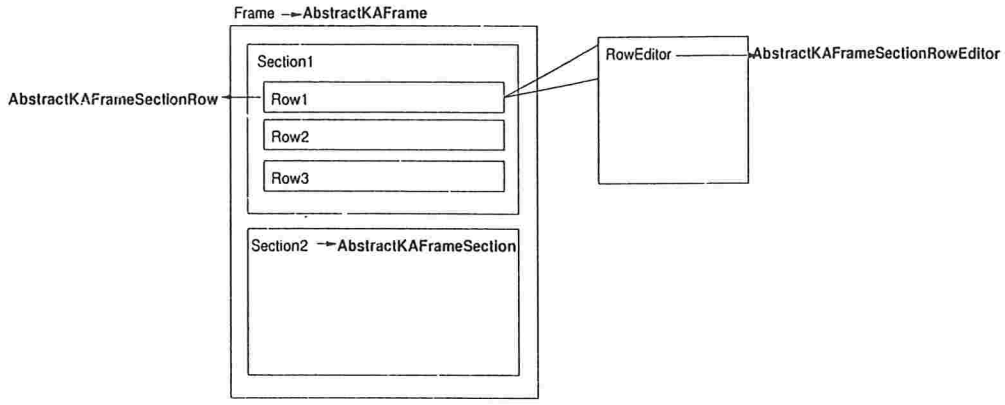

Figura 5.6: Esquema visual da lista usada na classe **KARuleDescriptionView**

No Protègè toda ontologia é criada com um conjunto padrão de ontologias importadas automáticamente. Para poder usar o nosso plugin, a ontologia ativa deve ser *convertida* para um AC. Esse processo de conversão consiste em copiar todos os axiomas de um AC previamente definido (Veja Anexo A). Isto inclui todas as ontologias que um AC deve importar, principalmente a definida na Seção 5.1. A classe **ConvertToKAWizard** inicia este processo mediante a seleção de uma ontologia que contenha uma instância de um AC. Depois da seleção, a classe **KALoader** tenta carregar o AC. Se ele tiver mesmo um AC [7], a classe **KAConverter** copia todos os axiomas desta ontologia na ontologia ativa no Protègè . Logo as classes que implementam os componentes visuais descritos na Figura 5.4 carregam os elementos correspondentes. Depois que a classe **KAConverter** acabou o processo, ela passa a instância do AC para a classe **KAInstanceManager** que implementa o padrão "Singleton" pois só é permitido ter uma instância de um AC no Protègè .

Aspecto Interno

A arquitetura geral do plugin implementa o padrão arquitetural Model-View-Controller (MVC). No nosso caso, o Model esta composto pelas classes Java que refletem o modelo ontológico explicado na Seção 5.1. A Figura 5.7 mostra estas classes.

A interface **KAObject** é o elemento central no modelo porque ela é implementada por todas as outras classes (e.g. "SyntaxRule"). Em geral, foi adotada a estratégia de fazer a implementação independente da interface (padrão de projeto "Bridge").

[7] Isto é, a ontologia contém uma instância da classe OWL *Knowledge_Artifact*

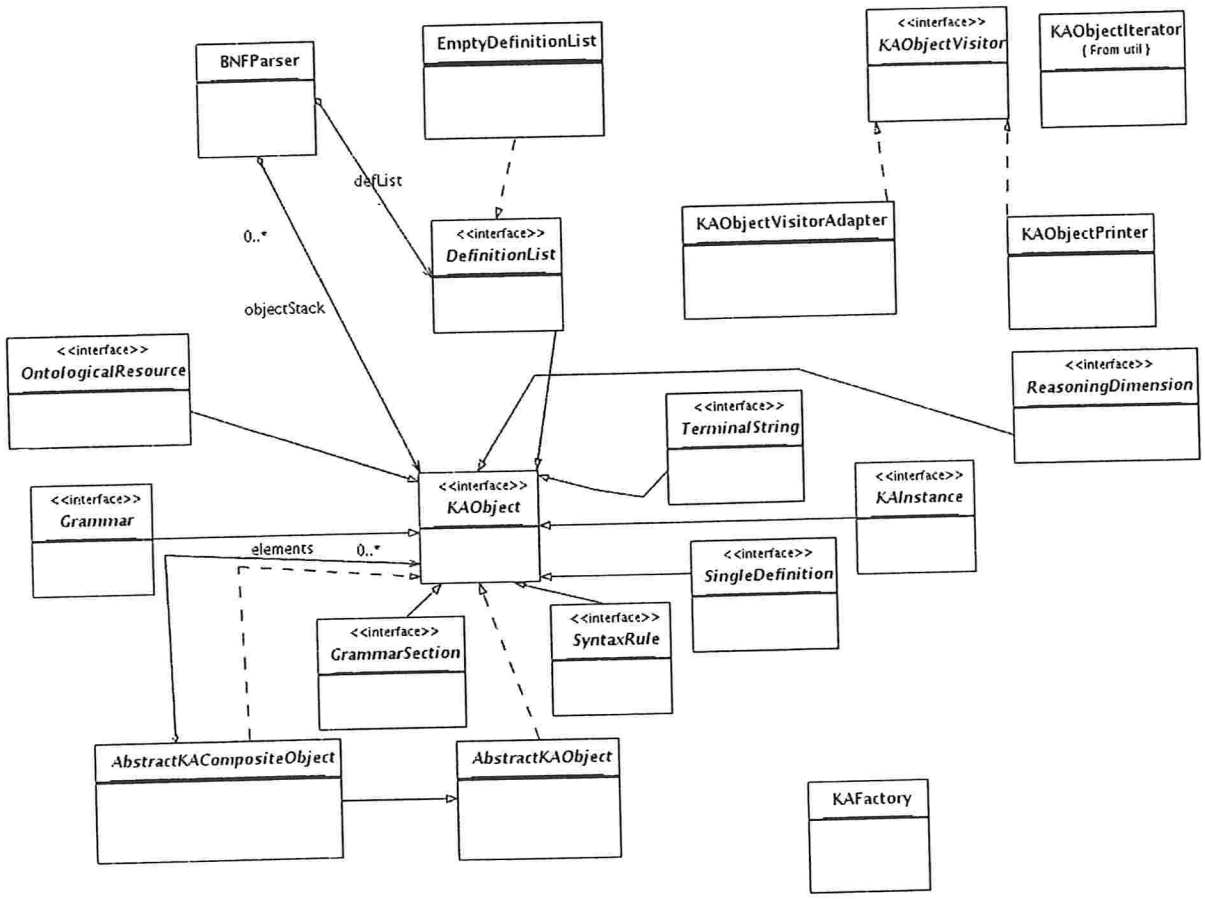

Figura 5.7: Diagrama das principais interfaces e classes do pacote **edu.usp.liamf.ka.model**

A maioria da funcionalidade das classes esta implementada em duas classes abstratas, **AbstractKAObject** e **AbstractKAObjectComposite** que junto com as classes filhas implementam o padrão "Composite" [8].

Junto com estas classes foram implementadas as interfaces **KAObjectIterator** e **KAObjectVisitor** para iterar ou visitar, respetivamente, objetos deste modelo.

[8] O exemplo de construção de uma regra sintática dado anteriormente (Figura 5.2) mostra como é necessário o uso deste padrão

Existem classes utilitárias como **KAFactory**, que é responsável pela criação de instâncias destas classes, assim como a classe **BNFParser** que implementa um pequeno parser para criar objetos deste modelo a partir de uma cadeia de carateres que cumpram com o formato de criação de regras sintática mostrado na Figura 5.2.

Capítulo 6

Discussão e Conclusões

Neste capítulo apresentamos algumas discussões com respeito aos limites da lógica usada na especificação do meta-modelo de AC. Discutimos como esta limitação pode ser superada mediante o uso de lógicas de ordem superior. Explicamos a seguir outras abordagens relacionadas com a nossa proposta de meta-modelo de AC. Específicamente, apresentamos os conceitos de "Knowledge Pattern" e "Prolog Skeleton" as suas diferenças, similaridades e possíveis contribuições para o nosso trabalho. Observamos alguns pontos faltantes neste trabalho e propomos as suas soluções como trabalhos futuros. Finalmente, mostramos algumas conclusões deste trabalho com respeito ao nosso método proposto, o meta-modelo de AC e a ferramenta desenvolvida.

6.1 Limites da Lógica Descritiva para AC

A criação do modelo baseado em AC usando linguagens da Web Semântica se mostrou difícil devido à falta de elementos na Lógica Descritiva para descrever um modelo desse tipo.

Fica claro, pelos níveis usados na especificação, que o meta-modelo de AC não pode ser usado para fazer consultas em motores de inferência clássicos como FACT ou Pellet. Existem porém alternativas como a linguagem F-Logic, que implementa Lógica de Ordem Superior [14], onde não teríamos o problema de copiar todos os axiomas de um modelo de AC para uma instância do modelo, como é feito na nossa ferramenta.

Podemos dar um exemplo: Suponha que tenha os axiomas que definem o meta-modelo de AC em F-Logic (usamos a sintaxe específica do motor de inferência Flora-2):

```
knowledge_artifact[grammar*=>grammar].
```

```
grammar[sections*=>grammar_section].
grammar_section[syntax_rules*=>syntax_rule,name*=>string].
syntax_rule[definition_list*=>definition_list,meta_id*=>string,isOption*=>boolean].
```

Para fins de exemplificação, definimos somente alguns axiomas para definir a estrutura geral do AC. Podemos, então, criar instâncias desta estrutura:

```
maintenance_and_repair:knowledge_artifact[grammar*->grammar1].
design_by_adaptation:knowledge_artifact[grammar*->grammar2].

grammar1:grammar[sections*->grammar_section1].
grammar2:grammar[sections*->grammar_section2].

car_repair:maintenance_and_repair.
```

A definição de instâncias de uma classe em Flora-2 segue o padrão "instância:classe". No nosso exemplo temos definidas duas instâncias da classe "knowledge_artifact": "maintenance_and_repair" e "design_by_adaptation". Flora-2 possui um conjunto de operadores para definir os atributos das classes. Estes operadores permitem definir a herança nestes atributos. Especificamente no nosso exemplo, temos definido que o atributo "grammar" seja herdado por todas as instâncias da classe onde foi definido.

Este é o caso da instância "car_repair", que representa o AC no domínio de reparo de motores de carro usando o AC "Maintenance and Repair".

É possível, ainda, fazer algumas consultas a este modelo para saber quais as instâncias do AC "Maintenance and Repair" e qual a gramática associada a estas instâncias:

```
?X:maintenance_and_repair,?X[grammar->?Y].

?X = car_repair
?Y = grammar1
```

Temos que o AC definido para o reparo de motores de carro herda a gramática definida no

AC "Maintenance and Repair". Finalmente, podemos obter todas as instâncias de AC mediante a seguinte consulta:

```
?X:knowledge_artifact.

?X = design_by_adaptation

?X = maintenance_and_repair
```

6.2 Outras abordagens

Apresentamos a seguir abordagens diferentes da idéia de AC, que podem levar a sistemas funcionalmente similares aos descritos nesse trabalho.

6.2.1 Knowledge Patterns

Em [16] são definidos padrões de construção de axiomas para bases de conhecimento. Estes padrões apresentam um conjunto de axiomas de alto nível, que são copiados na base de conhecimento quando aplicados. O modo de aplicação consiste em criar um morfismo, ou seja, um mapeamento entre os símbolos não-logicos do padrão para os nomes usados na base de conhecimento. Por exemplo, temos o seguinte padrão de conhecimento:

```
free-space(Container, F):-
    isa(Container,container),
    capacity(Container,C),
    occupied_space(Container,O),
    F is C - O.
```

Este padrão tem o nome de "Contenedor" e podemos aplicá-lo para dizer que um computador tem uma certa capacidade de memória livre. Esta aplicação pode ser definida mediante o seguinte morfismo:

```
container -> computer
capacity  -> ram_size
```

```
free_space -> available_ram
occupied_space -> occupied_ram
isa -> isa
```

O que resultaria na seguinte teoria:

```
available_ram(Container, F):-
    isa(Container,computer),
    ram_size(Container,C),
    occupied_ram(Container,O),
    F is C - O.
```

6.2.2 Prolog Skeletons

Em [46] são definidos outros tipos de padrões, conhecidos como Prolog Skeletons. Este padrões definem estruturas de dados (e.g. listas) junto com os procedimentos para operar sobre elas. Além disso, são definidas "técnicas" de programação sobre estes skeletons para produzir "extensões", ou seja, aplicações particulares de um skeleton para um propósito específico. Podemos colocar o exemplo de um skeleton para percorrer uma lista particionando o fluxo de controle em dois ramos de execução, baseados na relação de pertinência do elemento inicial da primeira lista na segunda:

```
traverse([X|Xs], Ys):-
  member(X,Ys), traverse(Xs,Ys).
traverse([X|Xs], Ys):-
  nonmember(X,Ys), traverse(Xs,Ys).
traverse([],Ys).
```

A técnica de união de duas listas pode fazer uso deste skeleton, criando uma extensão:

```
union([X|Xs],Ys,Us):-
  member(X,Ys),
  union(Xs,Ys,Us1),
  Us=Us1.
```

```
union([X|Xs],Ys,Us):-
  non_member(X,Ys),
  union(Xs,Ys,Us1),
  Us=[X|Us1].
union([],Ys,Us):-
  Us=Ys.
```

Estes skeletons, junto com os padrões anteriormente descritos, podem ser aplicados na teoria de AC como uma técnica mais granular no sentido que o seu âmbito de aplicação são bases de conhecimento específicas (e.g. ontologias). Estas técnicas poderiam ser implementadas na nossa ferramenta como um plugin adicional para sugerir ao Engenheiro de Conhecimento um certo tipo de padrão para uma problema específico de modelagem de conhecimento. Em geral, poderíamos fazer um paralelo entre padrões de software e de conhecimento como segue:

Padrão de software	Padrão de Conhecimento
Arquitetural (e.g MVC)	Artefato de Conhecimento (e.g. Design by Adaptation)
Padrão de Projeto (e.g. Observer)	Knowledge Patterns, Skeletons

6.3 Trabalhos Futuros

O nosso trabalho explorou somente o aspecto de modelagem de um AC no contexto do método proposto no Capítulo 4. O aspecto de geração ou teste de código fonte é o outro ponto importante a ser resolvido em um trabalho futuro.

Para resolver este aspecto temos opções interessantes de implementação, como por exemplo o framework Eclipse Equinox para definir um framework arquitetural para Sistemas baseados em AC baseando-nos no framework descrito na Seção 2.3. Este framework poderia definir pontos de extensão para todos os componentes a serem criados para uma aplicação especifica (e.g. Reparo de Carros).

Outro item a ser implementado é o componente gráfico que mostra e edita as sentenças baseadas nas regras gramaticais (Figura 5.4, classe **KASentencesView**). Do ponto de vista semântico, isto tem a ver com a criação de instâncias OWL a partir de texto, seguindo o formato definido pela regra gramatical.

Por outro lado, pode ser criado um plugin em Eclipse para gerar código fonte representando as sentenças da gramática e associar este código ao framework. Mais especificamente, estes objetos

podem ser gerados com base na classe Java "SyntaxRule" com métodos que devolvam o seu conteúdo sintático e semântico. Este último tem a ver com a recuperação das instâncias dos elementos OWL associados à regra.

Finalmente, o método de resolução de problemas do AC pode ter também um componente de software que gere um esquema de procedimentos a partir da descrição do fluxo de execução. Podemos pensar em um executor de especificações OWL-S especialmente implementado para a nossa especifição de entradas e saidas, que no nosso caso são instâncias de regras sintáticas.

6.4 Conclusões

A ferramenta aqui apresentada provê suporte parcial à prática de "Design Conjunto de AC", pois o Engenheiro de Conhecimento, junto com Agente Especialista, usa a ferramenta para acrescentar novas regras sintáticas e associá-las a elementos ontológicos. A interface gráfica enfatiza ao usuário se concentrar no modelo baseado em AC e não nos elementos ontológicos que fornecem a semântica. Isto seria complementado de melhor forma se o componente visual para criar instâncias às regras sintâticas fosse completado.

A prática de "Padronização de Modelagem" é suportada de forma natural, pois o Engenheiro, junto com o Especialista, define o vocabulário usado no domínio de aplicação do AC, que no caso de estudo seria o reparo de motores de carro.

De forma indireta, a ferramenta dá suporte à prática de "Refatoração de Dimensões", pois pode ser reutilizada uma ontologia (e.g. uma ontologia de motores de carro) para a definição de semântica das regras sintáticas. Poderia ser desenvolvido também um plugin para prover um método de refatoração de elementos ontológicos de forma que se possa copiá-los ou movê-los de uma ontologia para outra.

Quanto à interface gráfica, ela precisa ser projetada de uma forma melhor para exibir o modelo baseado em AC para os usuários. Isto deveria ser um item dentro de um projeto de aplicação real da ferramenta, para poder ter feedback dos usuários. A ontologia para modelos baseados em AC proposta no Capítulo 5 poderia também ser depurada com base nesse projeto.

De forma geral, é preciso testar a metodologia proposta no Capítulo 4 para permitir a sua evolução mediante a validação dos seus valores, princípios e práticas a partir da experiência em casos reais.

Por outro lado, foi identificada uma limitação da Lógica Descritiva para especificar modelos baseados em AC. Isto afetou a criação dos modelos em OWL na ferramenta desenvolvida. Uma

possível solução seria extender a lógica que esta por trás de OWL-DL para incluir axiomas que possam facilitar a criação de modelos e meta-modelos de AC.

Apêndice A

Utilização da Ferramenta

Neste capítulo mostramos um caso de uso da ferramenta proposta. Para poder testar a nossa ferramenta precisamos de um modelo de AC inicial. Para tal fim usamos o AC "Maintenance and Repair" cujo modelo foi criado usando o Protègè . Nele foram definidos elementos gramaticais que apontam a classes e propriedades OWL inexistentes na ontologia onde o AC é aplicado, mas que depois a ferramenta ajuda a criar.
Na seção A.1 mostramos a descrição do "Maintenance and Repair" usando o modelo ontológico descrito no capítulo anterior. Logo na seção A.2 mostramos a utilização da nossa ferramenta usando o modelo antes definido tentando nos aproximar do modelo baseado em AC para o reparo de motores de carro explicado no Capítulo 2.

A.1 Modelo do AC "Maintenance and Repair"

O AC "Maintenance and Repair" explicado no Capítulo 2 define um conjunto de regras gramaticais que sugerem conceitos ontológicos a serem criados pelo Engenheiro de Conhecimento.
A estrutura deste artefato esta dada pela estrutura mostrada na Figura A.1 e o seu método de resolução de problemas é mostrado na Figura A.2:

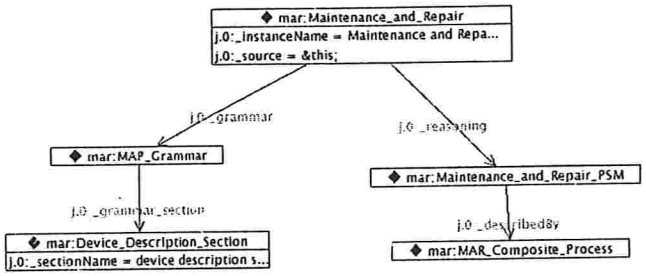

Figura A.1: Individuos OWL que constituem o AC **Maintenance and Repair**

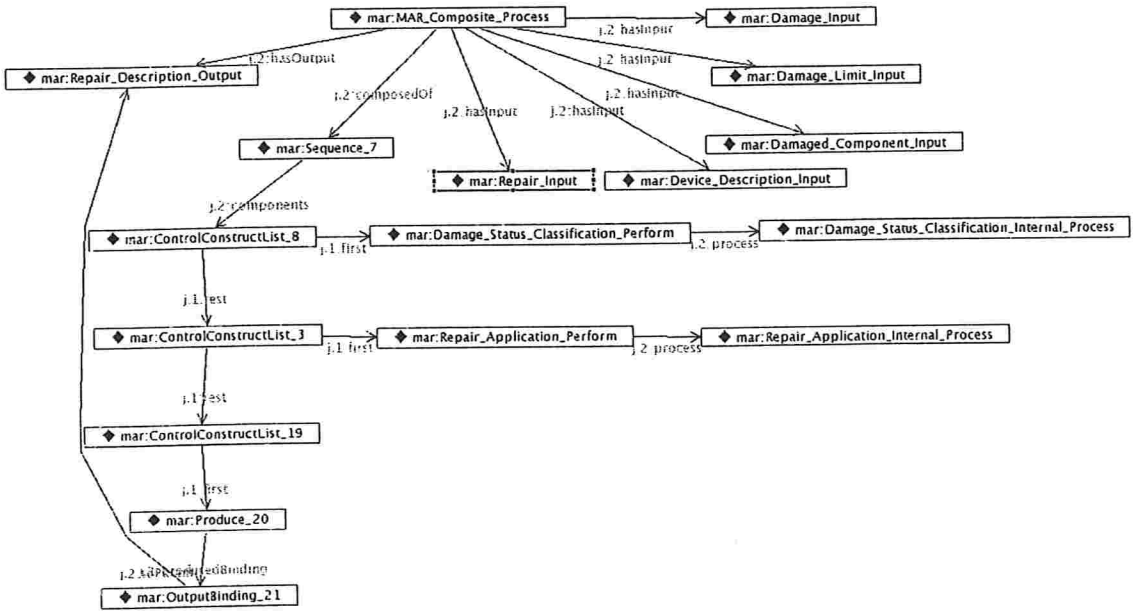

Figura A.2: Individuos OWL que constituem o método de resolução de problemas do AC **Maintenance and Repair**

A.2 Caso de uso

Implementamos o exemplo dado no capítulo 2 sobre o domínio de reparo de pistões de carros.

Começamos o processo mediante a criação de uma nova ontologia usando Protègè . Esta ontologia importa somente ontologias padrão que o mesmo Protègè insere. A Figura A.3 mostra a criação de

esta ontologia.

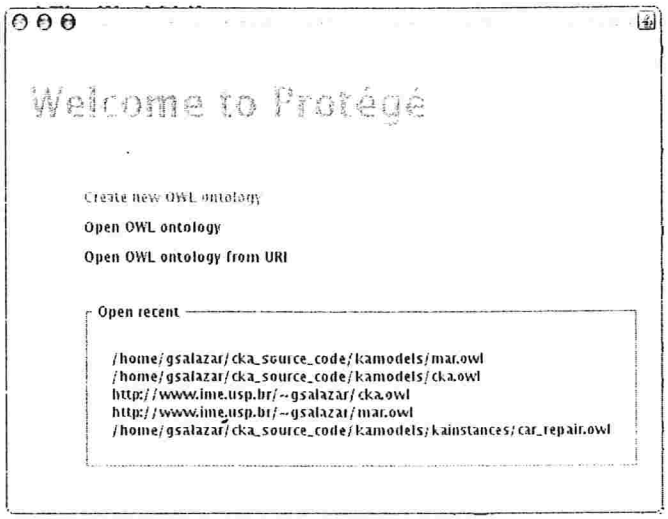

Figura A.3: Criação de uma ontologia no Protègè

O Protègè mostra automaticamente as vistas do nosso plugin sob a aba "Knowledge Artifact Grammar" que atualmente esta vazia. Neste ponto, precisamos criar uma nova instancia de um modelo de AC. O nosso procedimento de carregar o AC usa um motor de inferência para OWL-DL. Como mostra Figura A.4, isto é feito escolhendo um dos motores disponíveis em Protègè .

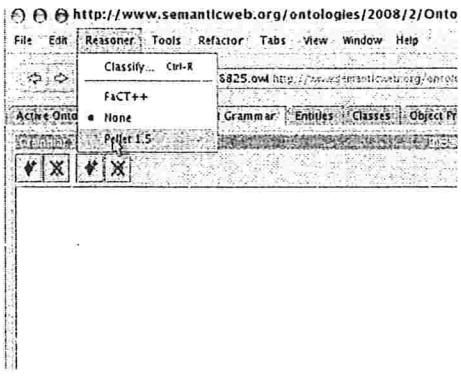

Figura A.4: Seleção de um motor de inferência no Protègè

Logo se inicia o Wizard de criação de uma instância de AC que encontra-se no menu "File→Convert ontology to Knowledge Artifact". A Figura A.5 mostra o primeiro passo do Wizard, onde a URL de uma ontologia deve ser fornecida para obter todas as instâncias de AC e poder escolher uma. Neste caso, escolhemos o AC "Maintenance and Repair" [1] . No segundo passo (Figura A.6) escolhemos um nome para a nossa instância de AC, que identifica o domínio onde este AC será aplicado. Seguindo o exemplo de pistões para carros, colocamos "Car Repair" para identificar o propósito desta instância.

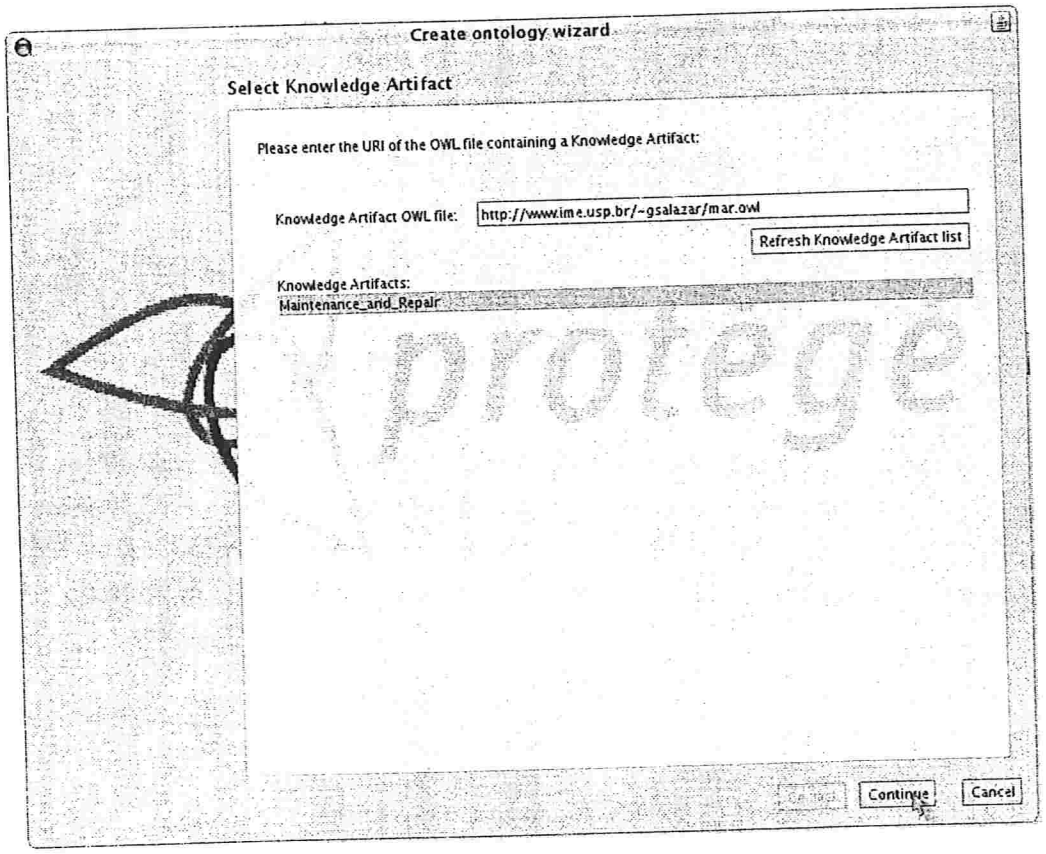

Figura A.5: Selecionando um AC a partir de um arquivo OWL

Depois de finalizado esse passo, o processo de carregar o modelo é iniciado mediante uma invocação ao método **loadKAInstanceFromOWLFile** da classe **KALoader**. Este retorna uma instância de **KAInstance**, que é repassada para a classe **KAInstanceManager** para gerenciá-la.

[1] A ontologia para este AC encontra-se em http://www.ime.usp.br/~gsalazar/mar.owl

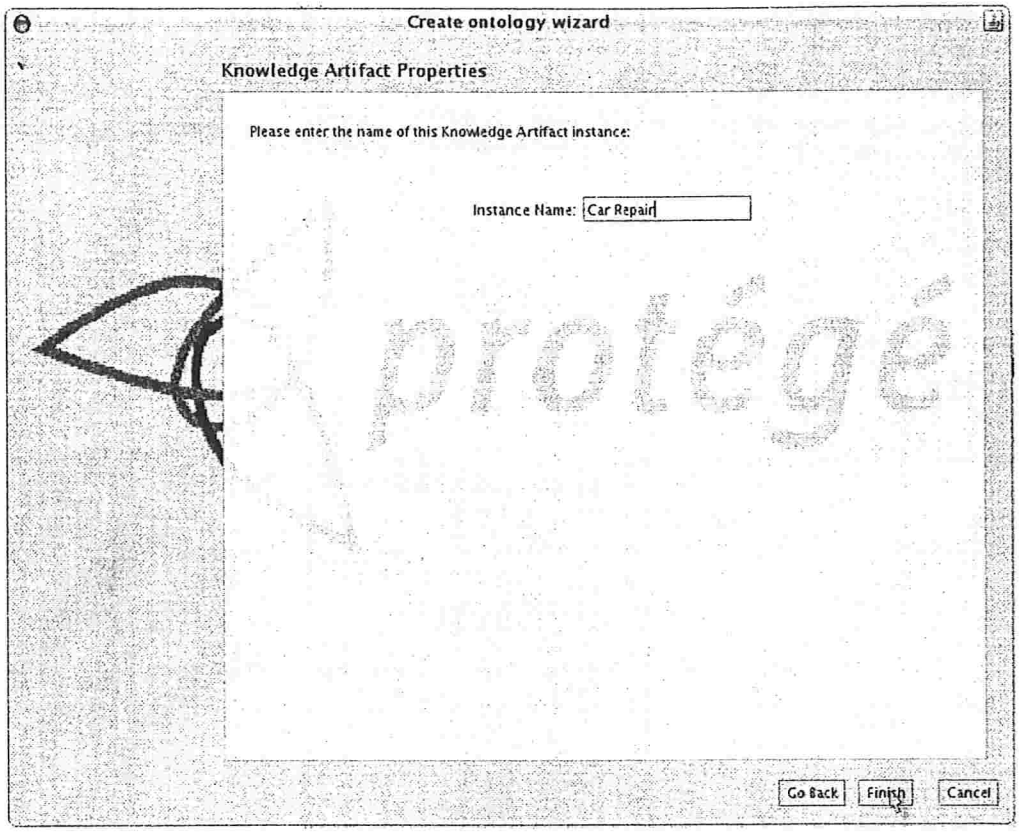

Figura A.6: Dando um nome para a instância concreta de AC

Uma vez carregada a instância do AC, a ferramenta mostra o esquema (Figura A.7) de janelas mostrado no capítulo anterior (Figura 5.4) com as sentenças carregadas.

Neste ponto, cada uma das regras gramaticais possui conteúdo semântico, ou seja, cada um dos elementos gramaticais que compõem a regra tem um apontador para um elemento OWL (e.g. classe), porém ,este pode não existir na ontologia atual. A ferramenta ajuda o Engenheiro de Conhecimento a criar estes elementos mediante o editor de regras.

Precisamos agora criar o modelo de AC para o caso de estudo explicado no Capítulo 2. Criamos as seguintes regras gramaticais, que fazem parte do domínio de reparo de motores de carro:

```
<cylinder arrange> ::= "v" | "flat" | "inline"
```

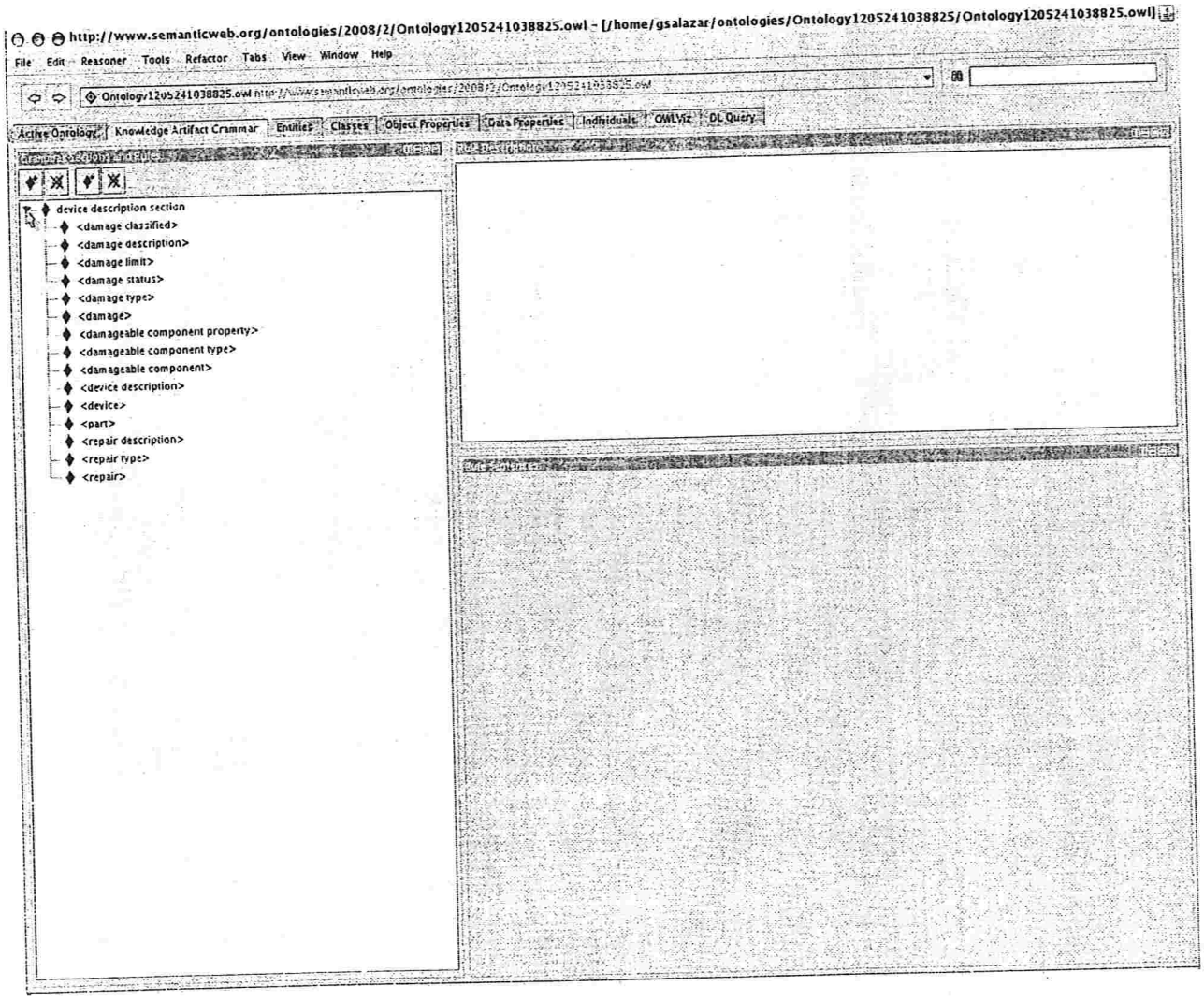

Figura A.7: Instância concreta carregada no nosso plugin

```
<cylinder bank> ::= "bank1" | "bank2"

<cylinder bank description> ::=
    <cylinder bank> " has " <cylinder arrange> " arrange" |
    <cylinder bank> " has " <number> "cylinders" |
    <cylinder bank> " is composed by " <cylinder>

<cylinder material> ::= "aluminum" | ...

<cylinder> ::= "cylinder1" | "cylinder2" | ...

<cylinder description> ::=
```

```
    <cylinder> " contains " <piston> |
    <cylinder> "has depth" <real number> |
    <cylinder> " has " <cylinder material>

<piston> ::= "piston1" | "piston2" | ...

<piston descripton> ::=
    <piston> " has " <piston material> " material" |
    "(" <piston> "," <piston component> ")"

<system> ::= "mm" | "cm"

<piston component> ::= "piston1head" | "piston1skirt" | ...

<piston material> ::= "aluminum" | "ceramic-aluminum"

<piston component type> ::= "piston head" | "piston skirt"

<piston component description> ::=
    "(" <piston component> "," <piston component type>, <piston component property> "," <amount> <system> ")"

<piston component property> ::= "thickness"

<mechanical> ::= "crack" | "hole" | ...
```

A Figura A.8 mostra a criação de regra <piston>. Uma vez criada a regra, temos que fornecer uma semântica para ela. De acordo com o nosso modelo, a semântica das sentenças BNF está dada pela semântica da classe OWL à qual ela esta associada. O processo todo é mostrado nas Figuras A.8, A.9 e A.10:

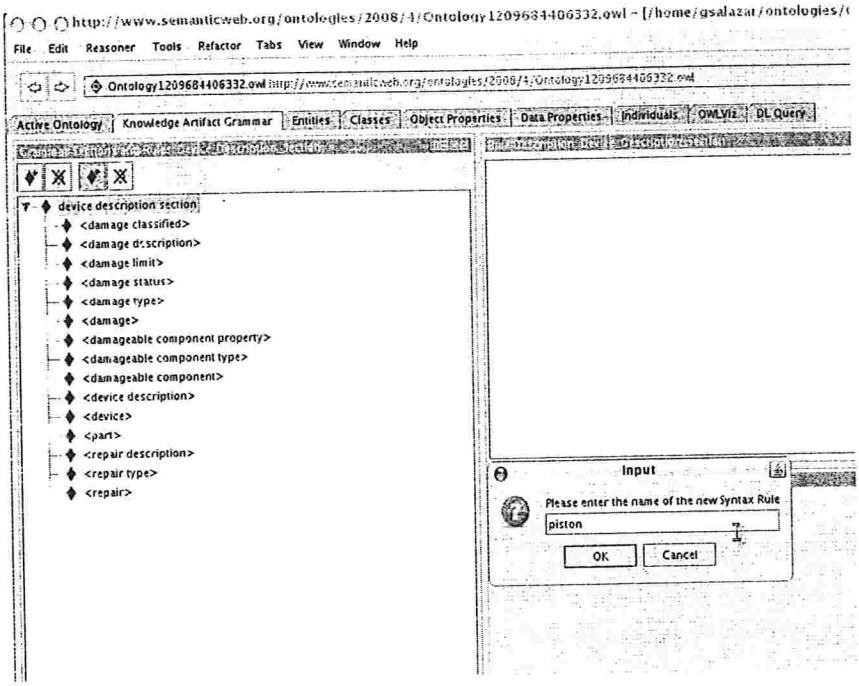

Figura A.8: Criação da regra "piston"

Aproveitamos a descrição deste último caso para criar a regra `<piston description>`, que possui uma construção mais complicada. A cada novo identificador de elemento gramatical podemos associar um elemento OWL (se for necessário), como mostra a Figura A.11.

No caso de ter-se um elemento gramatical que aponte para um tipo de dado primitivo, como um inteiro ou cadeia de carateres, depois que é dado o nome da regra (Figura A.12) a ferramenta pergunta se a regra está associada a um tipo de dados primitivo (Figura A.13). Se estiver, ela mostra todos os dados primitivos disponíveis no Protègè (Figura A.14).

Finalmente, mostramos na Figura A.15 todo o modelo completo com as regras dadas no caso de estudo.

A Figura A.16 mostra o diagrama das classes OWL criadas durante o processo de especificação do modelo baseado em AC para o domínio de reparo de carros [2].

[2] O arquivo OWL contendo este AC para o reparo de motores de carro encontra-se em http://www.ime.usp.br/gsalazar/CarRepair.owl

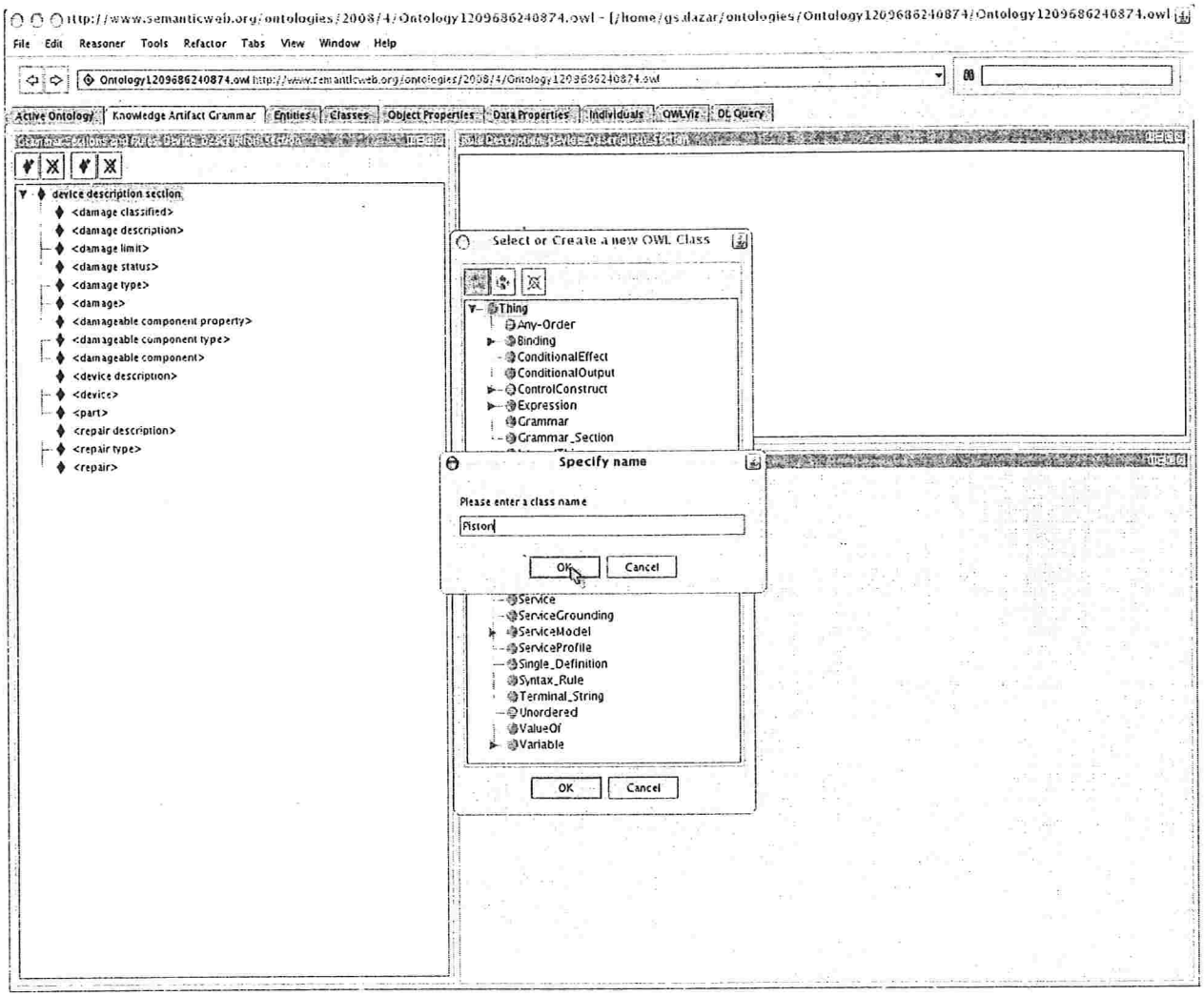

Figura A.9: Criando a classe OWL "Piston" associada à regra "piston"

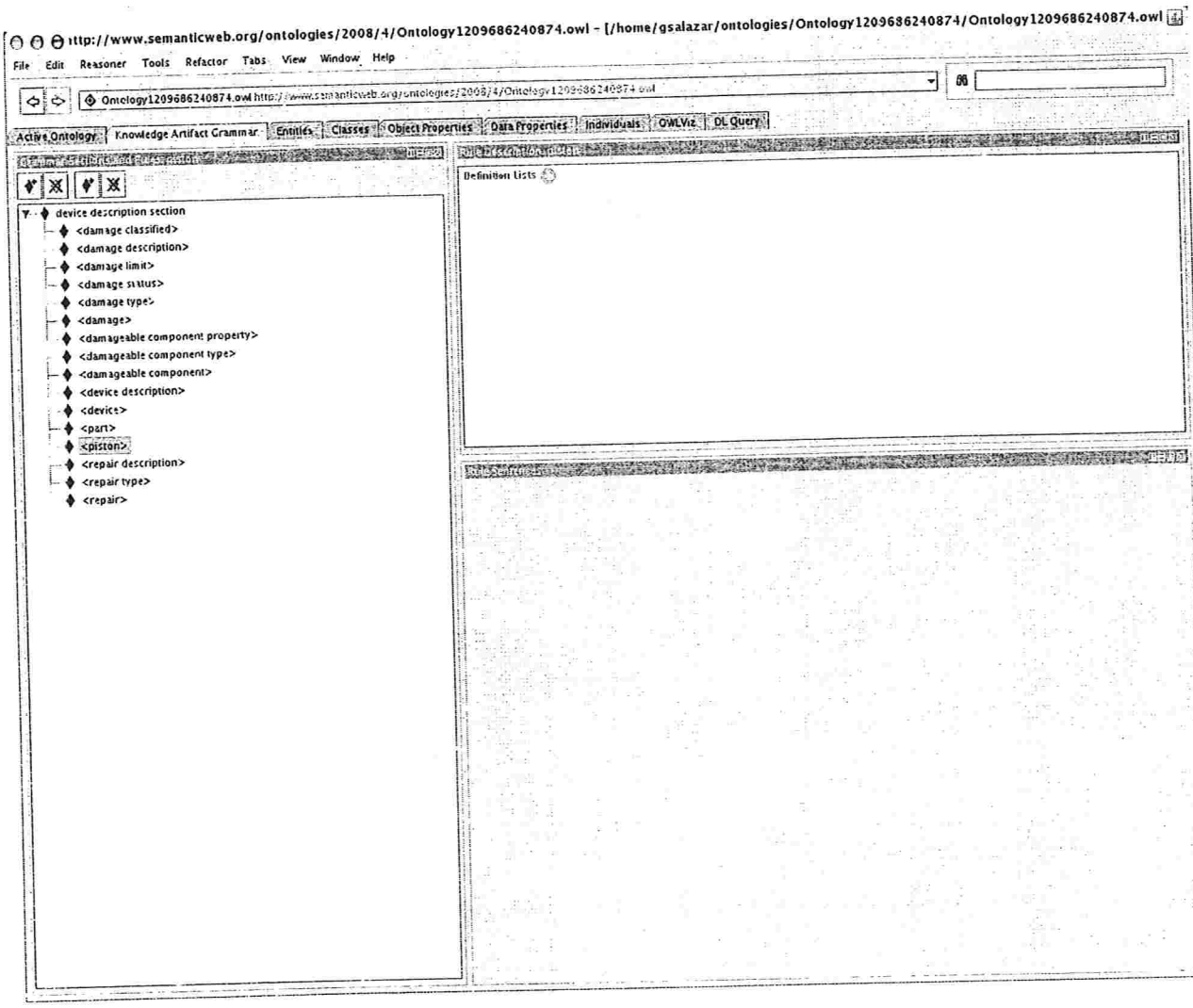

Figura A.10: Regra "piston" criada com uma lista vazia de definições

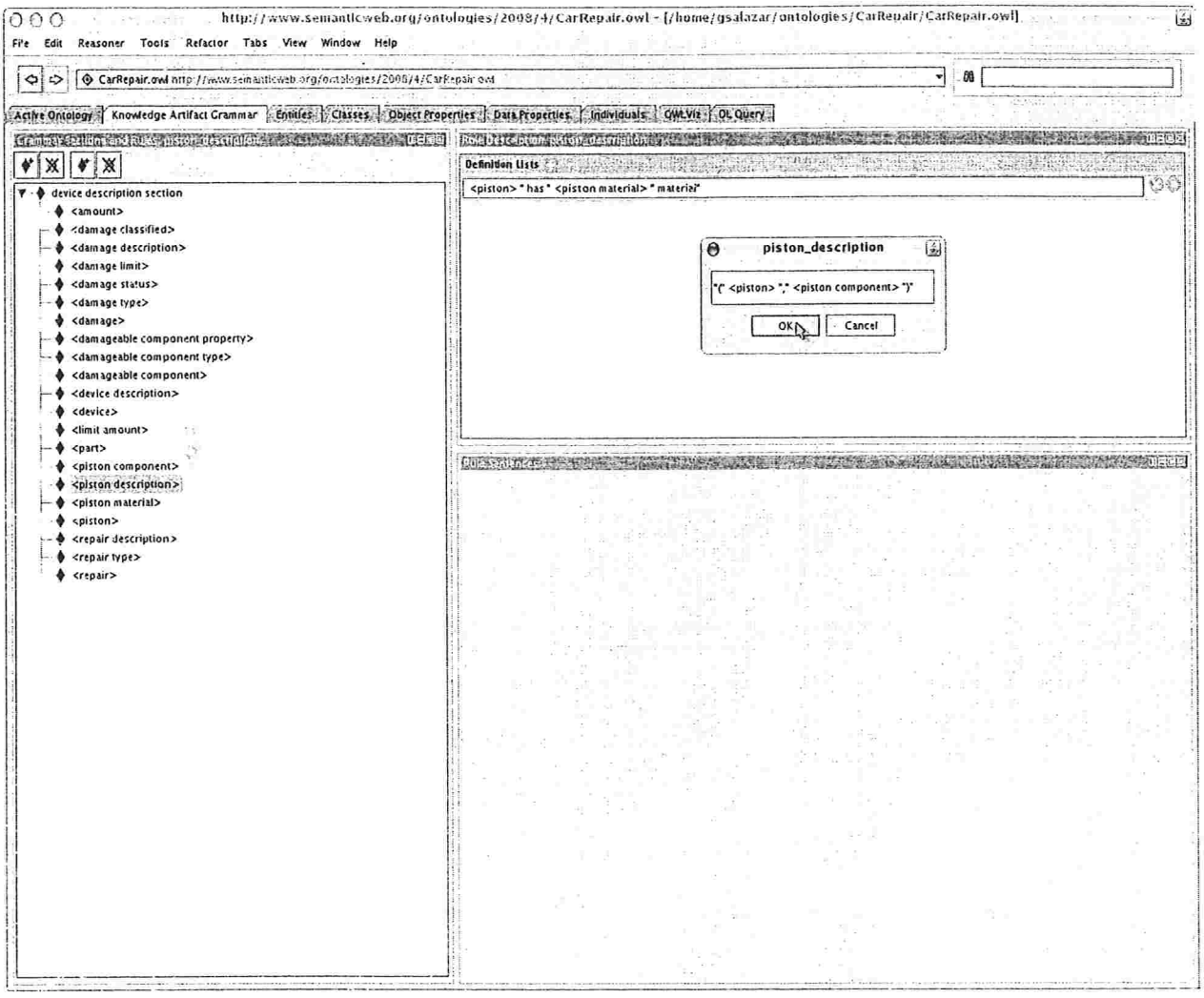

Figura A.11: Criando o conteúdo da regra "piston description"

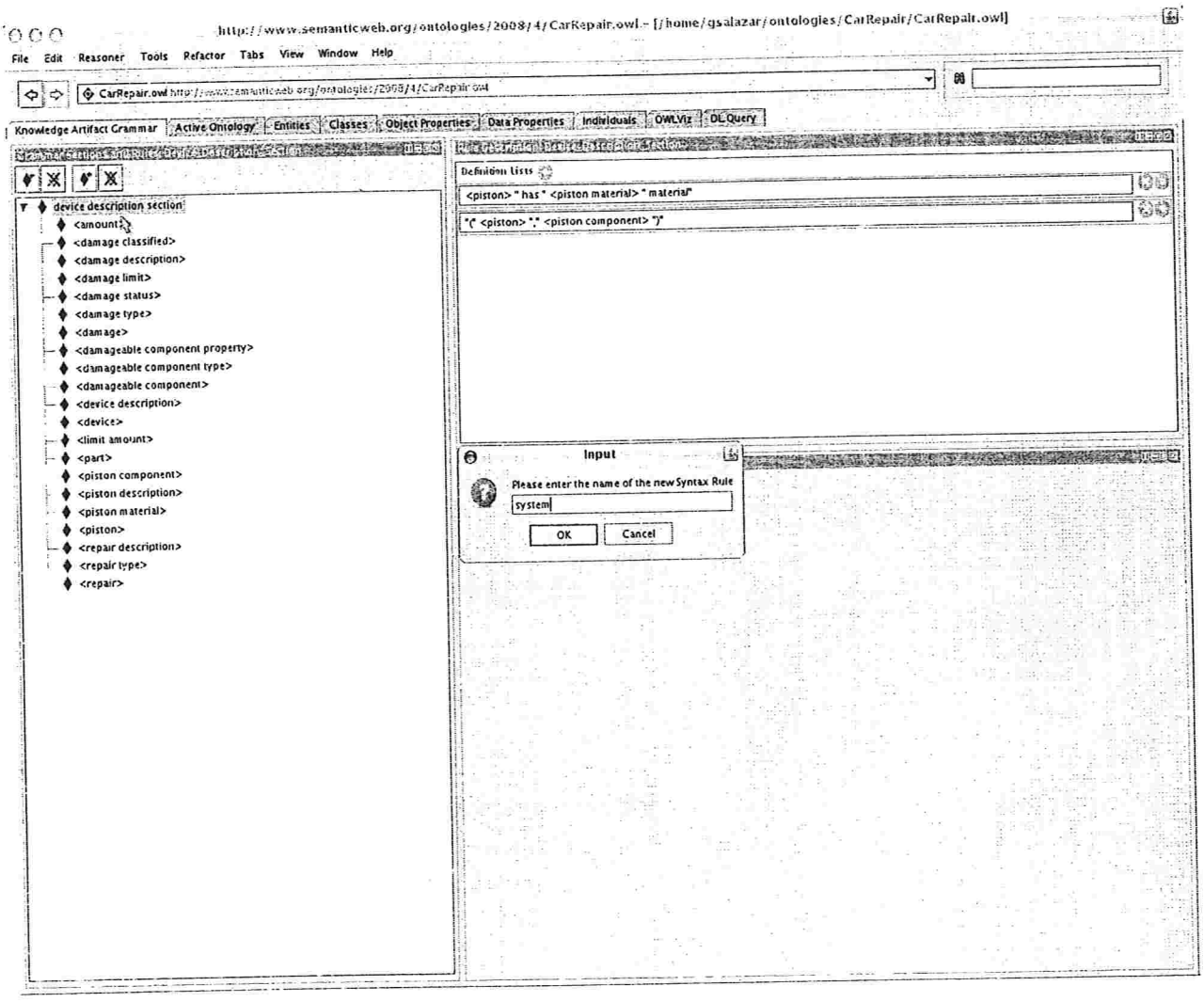

Figura A.12: Criando a regra "system"

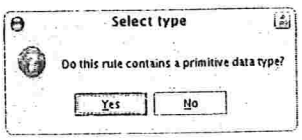

Figura A.13: Confirmando o tipo de entidade associada à regra "system"

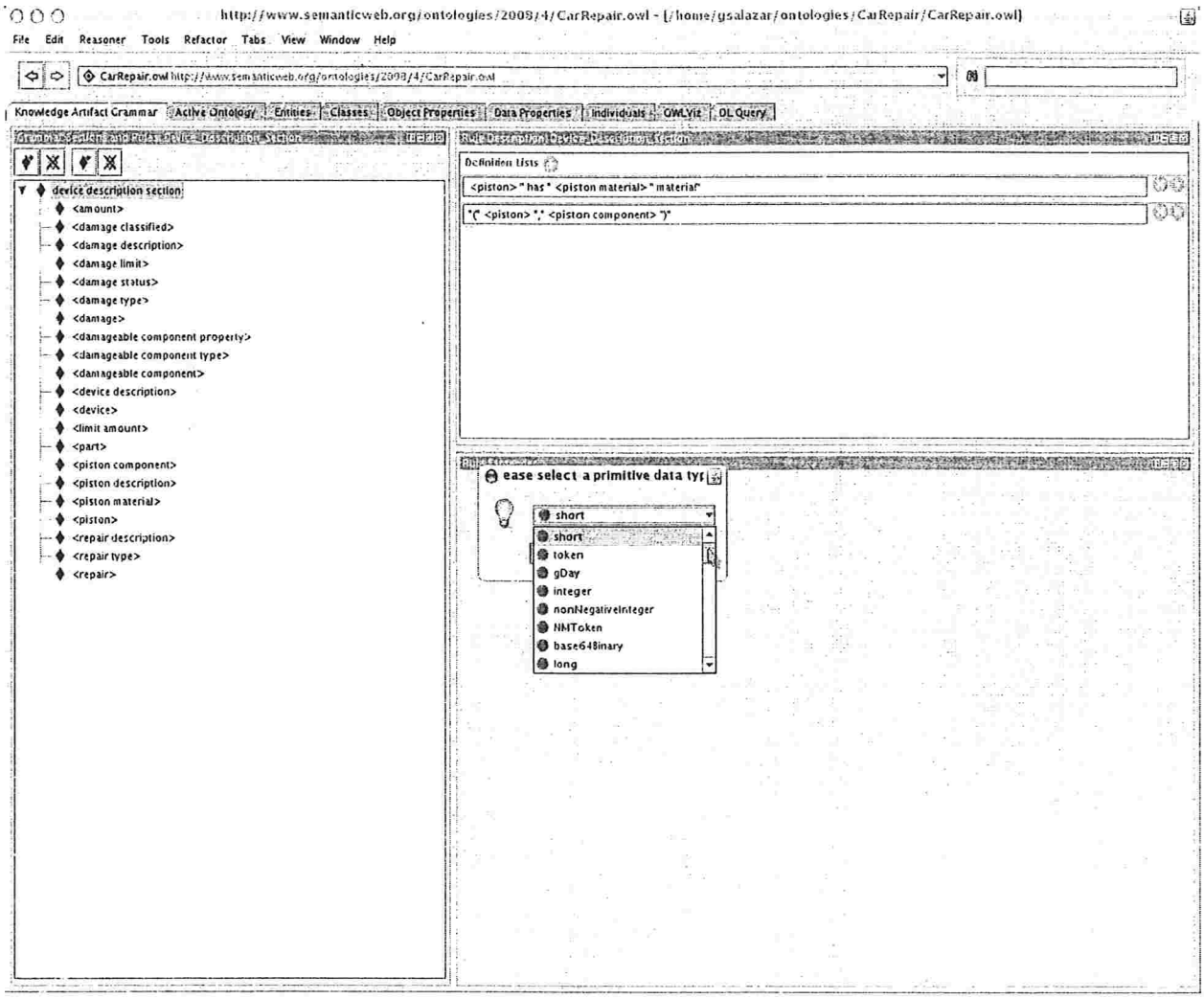

Figura A.14: Escolhendo o tipo de dado primitivo associado à regra "system"

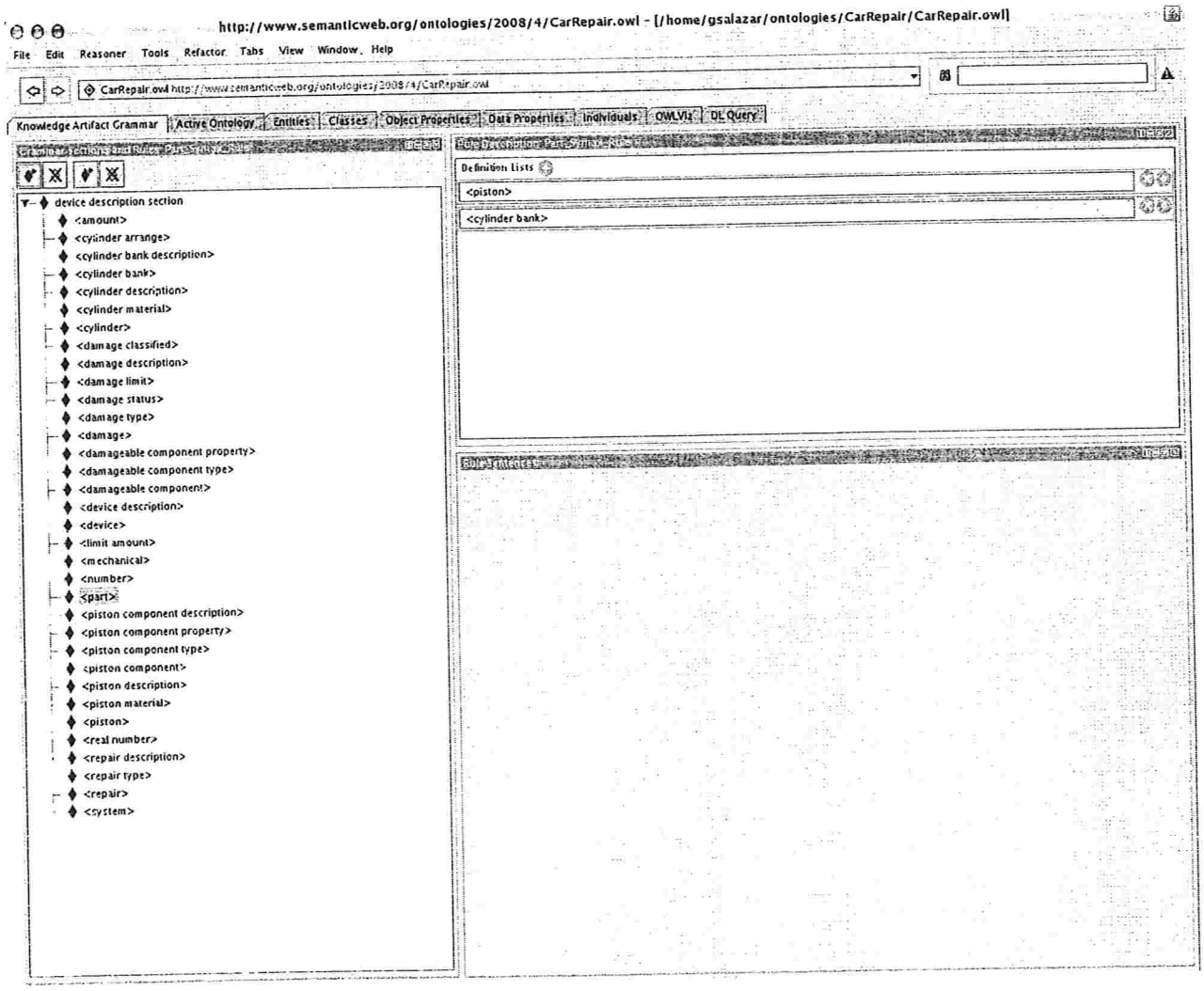

Figura A.15: O modelo completo do AC concreto para o domínio de reparo de motores

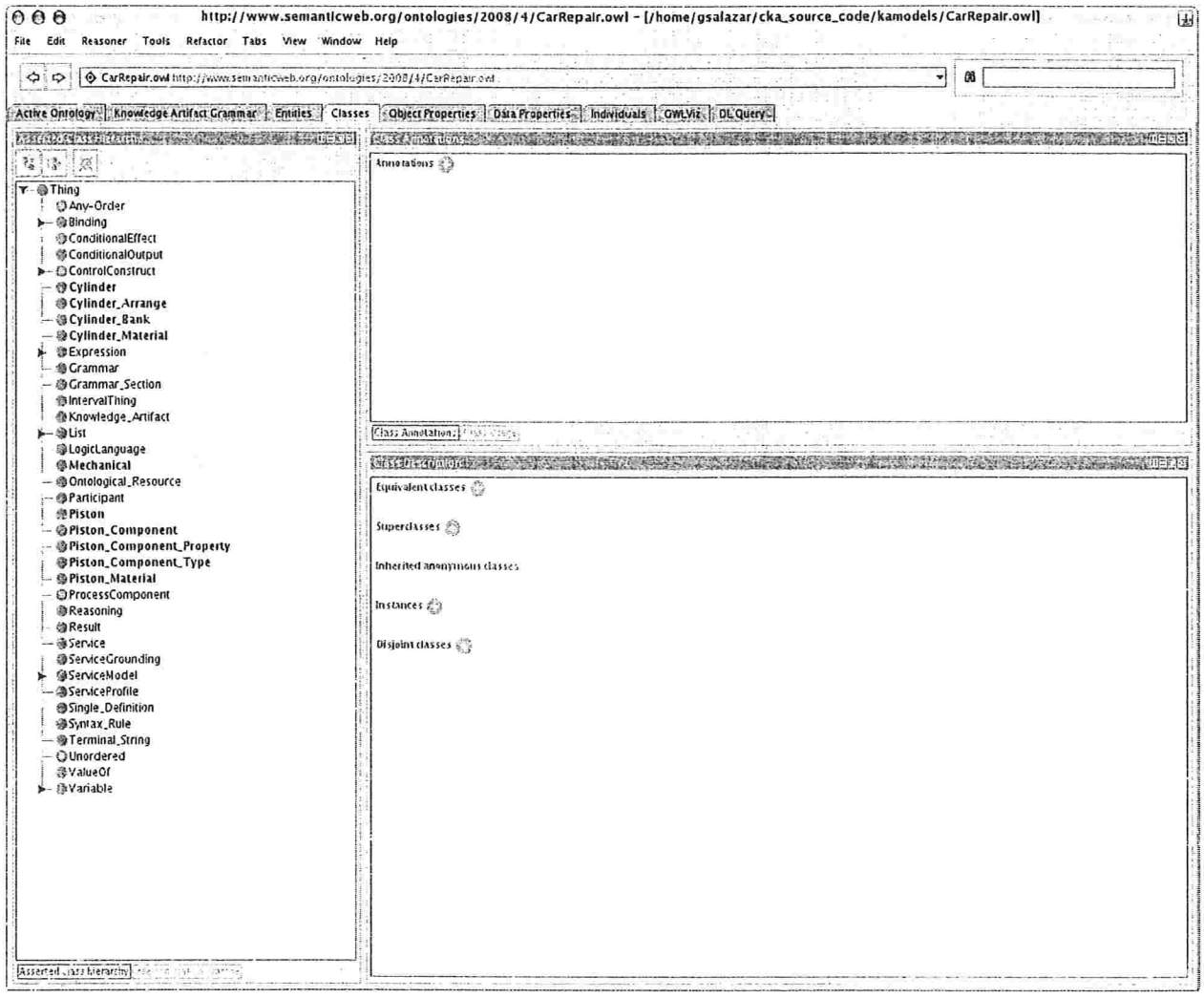

Figura A.16: Diagrama de classes OWL do AC concreto

Referências Bibliográficas

[1] OSGi Alliance. Open services gateway initiative. Disponível em http://www.osgi.org.

[2] J. Angele, D. Fensel, D. Landes, and R. Studer. Developing knowledge-based systems with MIKE. *Automated Software Engineering: An International Journal*, 5(4):389–418, October 1998.

[3] W.P. Appel and R Behr. Towards the theory of virtual organizations: A description of their formation and figure. *Arbeitspapiere wirtschaftsinformatik, Justus-Liebig-Universitt Gieen Fachbereich Wirtschaftswissenschaften,*, page 3, 1996.

[4] Soren Auer. *Towards Agile Knowledge Engineering: Methodology, Concepts and Applications.* PhD thesis, Universität Leipzig, 2006.

[5] Sören Auer, Sebastian Dietzold, and Thomas Riechert. Ontowiki - A tool for social, semantic collaboration. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, volume 4273 of *Lecture Notes in Computer Science*, pages 736–749. Springer, 2006.

[6] Sören Auer and Heinrich Herre. A versioning and evolution framework for RDF knowledge bases. In Irina Virbitskaite and Andrei Voronkov, editors, *Ershov Memorial Conference*, volume 4378 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2006.

[7] Sören Auer. 'The rapidowl methodology–towards agile knowledge engineering. In *WETICE*, pages 352–357. IEEE Computer Society, 2006.

[8] Stefania Bandini, Ettore Colombo, Gianluca Colombo, Fabio Sartori, and Carla Simone. *The role of knowledge artifacts in innovation management: the case of a chemical compound designer CoP*, pages 327–345. Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 2003.

[9] Stefania Bandini, Ettore Colombo, and Giuseppe Vizzari. The role of knowledge artifacts in knowledge maintenance. In Bhanu Prasad, editor, *Proceedings of the 2nd Indian International Conference on Artificial Intelligence, Pune, India, December 20-22, 2005*, pages 2795–2814. IICAI, 2005.

[10] Joachim Baumeister, Frank Puppe, and Dietmar Seipel. An agile process model for developing diagnostic knowledge systems. *KI*, 18(3):12–16, 2004.

[11] Borland. Together: Visual modeling for software architecture design, 2007. http://www.borland.com/us/products/together/index.html.

[12] E. Bozsak, Marc Ehrig, Siegfried Handschuh, Andreas Hotho, Alexander Maedche, Boris Motik, Daniel Oberle, Christoph Schmitz, Steffen Staab, Ljiljana Stojanovic, Nenad Stojanovic, Rudi Studer, Gerd Stumme, York Sure, Julien Tane, Raphael Volz, and Valentin Zacharias. Kaon - towards a large scale semantic web. In Kurt Bauknecht, A. Min Tjoa, and Gerald Quirchmayr, editors, *E-Commerce and Web Technologies, Third International Conference, EC-Web 2002, Aix-en-Provence, France, September 2-6, 2002, Proceedings*, volume 2455 of *LNCS*, pages 304–313. Springer, 2002. Disponível em http://kaon.semanticweb.org/.

[13] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: An architecture for storin gand querying RDF data and schema information. In Dieter Fensel, James A. Hendler, Henry Lieberman, and Wolfgang Wahlster, editors, *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential [outcome of a Dagstuhl seminar]*, pages 197–222. MIT Press, 2003. Disponível em http://www.openrdf.org.

[14] W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[15] Peter Clark, John Thompson, and Bruce Porter. Knowledge patterns. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 591–600, San Francisco, 2000. Morgan Kaufmann.

[16] Peter Clark, John Thompson, and Bruce Porter. Knowledge patterns. In R. Studer S. Staab, editor, *Handbook on Ontologies*. Springer, 2003.

[17] Ettore Colombo. *Knowledge Artifacts from an Artificial Intelligence Perspective*. PhD thesis, Scuola di Dottorato di Ricerca in Scienze, Università di Milano-Bicocca, 2005.

[18] Robin Cover. Xml and semantic transparency, 1998. Disponível em http://www.oasis-open.org/cover/xmlAndSemantic.html.

[19] W. Cunningham. Wiki design principles. Disponível em http://c2.com/cgi/wiki?WikiDesignPrinciples.

[20] Flávio Soares Corrêa da Silva and Jaume Agustí Cullell. *Knowledge Coordination*. John Wiley & Sons, 2003.

[21] Richard L. Daft. *Organization Theory and Design*. South-Western College Pub., 9 edition, January 2006.

[22] J P Delgrande and J Mylopoulos. Knowledge representation: features of knowledge. *Fundamentals of artificial intelligence: an advanced course*, pages 3–38, 1986.

[23] Zhi-Jun Ding, Junli Wang, and Chang-Jun Jiang. Semantic web service composition based on OWL-S. In *SKG*, page 98. IEEE Computer Society, 2005.

[24] D. Fensel, F. van Harmelen, I. Horrocks, D. L. McGuinness, and P. F. Patel-Schneider. Oil: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[25] Salazar-Torres G., Colombo E., Corrêa da Silva F., Noriega-Guerra C., and Bandini S. Design issues for knowledge artifacts. *Knowledge Based Systems*, 2008. Aceito para publicação.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[27] J. Giampapa, M. Paolucci, N. Srinivasan, and R. Vaculin. The OWL-S VM project, 2007. Disponível em http://projects.semwebcentral.org/projects/owl-s-vm/.

[28] V. Haarslev and R. Möoller. Racer: An owl reasoning agent for the semantic web. In *Proceedings of the International Workshop on Applications, Products and Services of Web-based Support Systems, in conjunction with the 2003 IEEE/WIC International Conference on Web Intelligence,Ê Halifax, Canada, October 13*, pages 91–95, 2003. Disponível em http://www.sts.tu-harburg.de/ r.f.moeller/racer.

[29] Steven Harnad. The symbol grounding problem. *Physica D.*, 42:335–346, 1990. Disponível em http://users.ecs.soton.ac.uk/ harnad/Papers/Harnad/harnad90.sgproblem.html.

[30] R. Hilpinen. ärtifact:̈ The Stanford Encyclopedia of Philosophy (fall 2004 Edition), Edward N. Zalta (ed.

[31] C. W. Holsapple and K. D. Joshi. Organizational knowledge resources. *Decis. Support Syst.*, 31(1):39–54, 2001.

[32] Holger Knublauch. *An Agile Development Methodology for Knowledge-Based Systems Including a Java Framework for Knowledge Modeling and Appropriate Tool Support.* PhD thesis, Fakultät für Informatik, Universität Ulm, 2002.

[33] Holger Knublauch. Semantic transparency for components and domain models, July 2002.

[34] Holger Knublauch, Martin Sedlmayr, and Thomas Rose. Design patterns for the implementation of constraints on javabeans, August 2000.

[35] Fernández-López M., Gómez-Pérez A., and Juristo. N. Methontology: from ontological art towards ontological engineering. In *Symposium on Ontological Engineering of AAAI*, 1997.

[36] Giorgio De Michelis. *Aperto, moteplice, continuo.* Dunod, Milano, 1998.

[37] Sun Microsystems. Javabeans specification. http://java.sun.com/products/javabeans/docs/spec.html.

[38] Martin J. O'Connor, Holger Knublauch, Samson W. Tu, Benjamin N. Grosof, Mike Dean, William E. Grosso, and Mark A. Musen. Supporting rule system interoperability on the semantic web with SWRL. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6-10, 2005, Proceedings*, volume 3729 of *Lecture Notes in Computer Science*, pages 974–986. Springer, 2005.

[39] Werner Rammert, Michael Schlese, Gerald Wagner, Josef Wehner, and Rüdiger Weingarten. *Wissensmaschinen: Soziale Konstruktion eines technischen Mediums. Das Beispiel Expertensysteme.* Campus Verlag, Frankfurt, Germany, 1998.

[40] Gavriel Salomon. *Distributed cognitions: Psychological and educational considerations.* Cambridge University Press, Cambridge, UK, 1993.

[41] Bernd Schmidt. *Die Modellierung menschlichen Verhaltens.* SCS-Europe Publishing House, Ghent, Belgium, 2000.

[42] Guus Schreiber, Bob J. Wielinga, Robert de Hoog, Hans Akkermans, and Walter Van de Velde. CommonKADS: A comprehensive methodology for KBS development. *IEEE Expert*, 9(6):28–37, 1994.

[43] E. Sirin, J. A. Hendler, and B. Parsia. Interactive Composition of Semantic Web Services. In *WWW (Posters)*, 2003.

[44] Evren Sirin and Bijan Parsia. Pellet: An OWL DL reasoner. In Volker Haarslev and Ralf Möller, editors, *Proceedings of the 2004 International Workshop on Description Logics (DL2004), Whistler, British Columbia, Canada, June 6-8, 2004*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[45] Mark Stefik. *Introduction to Knowledge-Systems.* Morgan Kaufmann, 1996.

[46] Leon Sterling and Marc Kirschenbaum. Applying techniques to skeletons. In *ICLP Workshop on Construction of Logic Programs*, pages 127–140, 1991.

[47] Rudi Studer, Dieter Fensel, and V. Richard Benjamins. Knowledge engineering: Principles and methods. *Data & Knowledge Engineering*, 25:161–197, 1998.

[48] Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke. Ontoedit: Collaborative ontology development for the semantic web. In *Proceedings of the 1st International Semantic Web Conference (ISWC2002)*, Sardinia, Italia, June 9-12th 2002. Springer.

[49] York Sure, Steffen Staab, and Rudi Studer. On-to-knowledge methodology (OTKM). In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 117–132. Springer, 2004.

[50] Laurence Prusak Thomas H. Davenport. *Working Knowledge.* Hardvard Business School Press., 2 edition, May 200.

[51] Amrit Tiwana. *The Knowledge Management Toolkit: Practical Techniques for Building a Knowledge Managament System.* Prentice Hall, 1999.

[52] Etienne Wenger. *Communities of Practice: Learning, Meaning and Identity.* Cambridge University Press., Cambridge, 1998.

[53] Etienne Wenger. Knowledge management as a doughnut: Shaping your knowledge strategy through communities of practice. *Ivey Business Journal*, page 6, January/February 2004.

[54] Terry Winograd. *Bringing Design to Software.* Addison-Wesley, 1996.

[55] Terry Winograd. Shifting viewpoints: Artificial intelligence and human-computer interaction. *Artificial Intelligence*, 170:1256–1258, 2006.

[56] World-Wide Web Consortium. *Resource Description Framework (RDF)*, 1998. Disponível em http://www.w3.org/RDF/.

[57] Guizhen Yang, Michael Kifer, and Chang Zhao. Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In *Second International Conference on Ontologies, Databases and Applications of Semantics (ODBASE)*, Catania, Sicily, Italy, November 2003. Disponível em http://flora.sourceforge.net.