UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE CIÊNCIA E TECNOLOGIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

KILDARE ALVES DA SILVEIRA

REENGENHARIA E EVOLUÇÃO DA FERRAMENTA KSE PARA APRIMORAR A CAPTURA DE DESIGN RATIONALE

Rio das Ostras-RJ

2016

UFF – UNIVERSIDADE FEDERAL FLUMINENSE

KILDARE ALVES DA SILVEIRA

REENGENHARIA E EVOLUÇÃO DA FERRAMENTA KSE PARA APRIMORAR A CAPTURA DE DESIGN RATIONALE

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação do Instituto de Ciência e Tecnologia da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel. Área de concentração: Engenharia de Software.

ORIENTADORA: Prof. D.Sc. ADRIANA PEREIRA DE MEDEIROS

Rio das Ostras-RJ

2016

Kildare Alves da Silveira

REENGENHARIA E EVOLUÇÃO DA FERRAMENTA KSE PARA APRIMORAR A CAPTURA DE DESIGN RATIONALE

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação do Instituto de Ciência e Tecnologia da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel. Área de concentração: Engenharia de Software.

Aprovada em 20 de outubro de 2016.

BANCA EXAMINADORA

_____

Prof. D.Sc. ADRIANA PEREIRA DE MEDEIROS - Orientador

UFF

_____

Prof. D.Sc. CARLOS BAZILIO MARTINS

UFF

_____

Prof. D.Sc. EDUARDO MARQUES

UFF

Rio das Ostras-RJ

2016

Agradecimentos

Agradeço à minha família, meus pais e meus irmãos pela paciência e compreensão das dificuldades na realização deste trabalho.

Agradeço à minha parceira de todas as horas, Gabriela, sem a qual este trabalho não teria sido realizado.

Agradeço à minha orientadora pelas horas dedicadas na melhoria deste trabalho.

Lista de Figuras

Lista de Códigos

Lista de Algoritmos

Sumário

Resumo

A ferramenta KSE (*Kuaba Software Engineering*) utiliza a abordagem Kuaba para apoiar a captura de *Design Rationale* durante a modelagem de software. *Design Rationale* pode ser descrito como a forma de representação de todo o raciocínio envolvido no design de um artefato. Atualmente a ferramenta KSE apresenta erros na representação das decisões de design tomadas pelo projetista e não apoia a captura de justificativas para essas decisões. Neste trabalho é apresentada a descrição de um processo de reengenharia de software aplicado na KSE para corrigir a representação de decisões. Também é apresentada uma proposta de melhoria à ferramenta, através da implementação de uma evolução da ontologia Kuaba para a identificação e representação de soluções de design. Nesta proposta, a KSE é estendida para capturar e armazenar justificativas para as decisões tomadas com uma abordagem menos intrusiva ao usuário.

Palavras-Chave: *Design Rationale*. Kuaba. Engenharia de Software.

Abstract

KSE is a CASE (Computer-Aided Software Engineering) tool, which uses the Kuaba approach to capture Design Rationale during software design. Design Rationale may be described as tool for representing the reasoning involved in the design of an artifact. This tool currently contains errors in the representations of the design decisions taken by the designer and does not support justifications for these decisions. This work describes a software reengineering process applied in the KSE to correct the representation of the decisions. In addition, this work describes a proposed evolution to this tool, through the implementation of a new element defined for the Kuaba ontology for representing design solutions. In this proposed evolution, the tool is extended to support the capture and representation of justifications for the decisions taken in the designing process with a less intrusive user interface.

Keywords: Design Rationale. Kuaba. Software Engineering.

Capítulo 1

Introdução

Historicamente, na literatura, as ferramentas de modelagem de software sempre possuíram o foco na captura do resultado de um processo de modelagem. O conhecimento envolvido na criação dos modelos não costuma ser levado em consideração, fazendo com que fatos e considerações relevantes para as escolhas das soluções de modelagem não sejam registradas durante este processo. A ferramenta KSE (*Kuaba Software Engineering*) [Nunes & Medeiros 2009] foi desenvolvida com o intuito de suprir essa necessidade de apoiar a modelagem de software e a captura do conhecimento utilizado nessa modelagem, denominado *Design Rationale*.

Design Rationale armazena o conhecimento empregado durante o design de um artefato. Este conhecimento consiste dos problemas de design tratados, as alternativas de solução para esses problemas, os argumentos contra e a favor de cada alternativa e as decisões tomadas [Lee 1997].

A KSE é uma extensão da ferramenta CASE (*Computer Aided Software Engineering*) de código aberto ArgoUML [ArgoUML 2008], que inclui um subsistema desenvolvido para apoiar a captura, representação e o processamento de *Design Rationale* utilizando a abordagem Kuaba [Medeiros 2006]. De acordo com essa abordagem, a ferramenta captura durante um processo de modelagem as modificações que o projetista realiza na área de desenho dos diagramas, sejam elas adições, remoções ou edições de elementos, e obtém, no instante em que tais modificações são realizadas, argumentos contra ou a favor do uso dos elementos do metamodelo UML. Com estas informações, a KSE é capaz de representar em tempo real o grafo de *Design Rationale* de acordo com a abordagem Kuaba.

Neste trabalho serão apresentadas algumas mudanças realizadas no subsistema Kuaba, com o objetivo de melhorar a captura de *Design Rationale*. Em sua última versão, a ferramenta KSE apresenta problemas na realização da captura automática das decisões tomadas pelos projetistas. Durante a modelagem de um diagrama, o projetista pode fazer alterações que geram decisões aceitas e rejeitadas. A KSE não processa essas alterações corretamente, gerando inconsistências nos elementos que compõem o *Design Rationale* e

ocasionando uma incompatibilidade entre o resultado obtido no diagrama e o registro de *Design Rationale*. Por isso, se faz necessário a realização de um processo de reengenharia na estrutura do subsistema Kuaba da KSE com o objetivo de solucionar esses problemas.Para correção destes problemas, serão aplicadas técnicas de reengenharia de software descritas em [Demeyer & Ducasse & Nierstraz 2008], com o intuito de obter uma maior compreensão da estrutura e funcionamento do subsistema Kuaba.

Além desses problemas, a ferramenta KSE não possui atualmente suporte à captura de justificativas para as decisões registradas pela ferramenta, como previsto pela abordagem Kuaba. Este trabalho também apresenta uma evolução da KSE, incluindo uma funcionalidade para apoiar a captura dessas justificativas. Esta funcionalidade implementa a modificação feita no vocabulário da ontologia Kuaba proposta em [Avila e Medeiros 2014] para tornar a captura e a representação dessas justificativas mais efetiva na KSE.

O restante desse trabalho está organizado da seguinte forma: no capítulo 2 é feita uma apresentação dos principais fundamentos aplicados, sendo estes, a definição da ontologia Kuaba, a estrutura da ferramenta KSE e da linguagem OWL, responsável por descrever formalmente a ontologia, e os conceitos de reengenharia de software aplicados na melhoria da ferramenta. Em sequência, o capítulo 3 apresenta uma descrição da ferramenta KSE. No capítulo 4, são descritas as melhorias que foram realizadas na estrutura da KSE e os aprimoramentos nos padrões de projeto utilizados, alinhados aos conceitos de reengenharia de software que foram descritos. No capítulo 5, descreve-se uma proposta de evolução da ferramenta, com a inclusão de um novo elemento na ontologia Kuaba para representar soluções de design. O capítulo 6 apresenta alguns testes realizados na ferramenta KSE para verificar os resultados obtidos. Por fim, são apresentados a conclusão e possíveis trabalhos futuros.

Capítulo 2

Fundamentos

2.1 Design Rationale

O processo de modelagem de software costuma ser auxiliado por ferramentas de software que otimizam e aperfeiçoam a geração e representação dos artefatos. Porém, nem sempre é fácil entender, a partir de modelos já existentes, quais os principais fatores que motivaram o projetista a gerar estes artefatos de um jeito específico.

Design Rationale é uma forma de descrever o conhecimento que foi aplicado durante a construção de um artefato. Essa descrição geralmente inclui os problemas de design tratados, as alternativas de solução consideradas, os argumentos prós e contras essas alternativas e as decisões tomadas. Uma representação adequada de *Design Rationale* permite que um projetista compreenda as decisões de projeto que foram tomadas para um determinado artefato. Estas informações são úteis para embasar o projetista na compreensão do artefato e na tomada de decisões sobre possíveis modificações. Sendo assim, o *Design Rationale* pode ser utilizado para compartilhar o conhecimento aplicado na solução de problemas de design, baseando-se nos artefatos gerados.

No processo de desenvolvimento de software, a geração de documentação de software nem sempre é priorizada. Devido a este fato, com o passar do tempo o conhecimento pode ser perdido. Por exemplo, uma pessoa pode não se lembrar de todos os aspectos que a levaram a gerar um artefato de uma maneira específica, ou esta pessoa responsável por gerar o artefato pode sair da organização. Fatos como estes podem acontecer e gerar prejuízos à organização. Assim, um procedimento automatizado para o registro de *Design Rationale* pode ser um facilitador para armazenamento de conhecimento e futuro compartilhamento.

2.2 A Abordagem Kuaba

A abordagem Kuaba consiste em utilizar os metamodelos de design como meio para realizar o registro semiautomático de *Design Rationale*, permitindo que o projetista realize este registro enquanto realiza a tarefa de modelagem. Esta abordagem busca fornecer ao projetista um processo mais natural na captura do *Design Rationale*, minimizando o tempo gasto em sua descrição e aproveitando a vantagem de realizar esta descrição enquanto realiza

a modelagem. Isto evita possíveis perdas de conhecimento que poderiam ocorrer, caso esta tarefa fosse realizada no futuro.

A abordagem Kuaba utiliza uma ontologia de mesmo nome para a representação de *Design Rationale*. Conforme descrito em (Guarino,2009), em tradução livre, “ontologias computacionais são meios de modelar formalmente a estrutura de um sistema, ou seja, as entidades relevantes e as relações que emergem das suas observações e que são úteis para os nossos propósitos”. No contexto deste pensamento, a formalização de um *Design Rationale* precisaria, necessariamente, se apoiar em uma descrição de entidades e relações que são pertencentes a este domínio.

A ontologia Kuaba surge como uma proposta de solução para o problema levantado no parágrafo anterior. Conforme descrito em (Medeiros 2006): ‘'Nosso objetivo ao propor esta ontologia é fornecer um vocabulário que permita atribuir semântica ao conteúdo de *Design Rationale* registrado, e definir um conjunto de regras que possibilite a realização de inferências e operações computáveis para apoiar o uso de *Design Rationale*".

A Figura 2.1 apresenta os elementos do vocabulário da ontologia Kuaba, em notação similar à UML para facilitar a visualização.

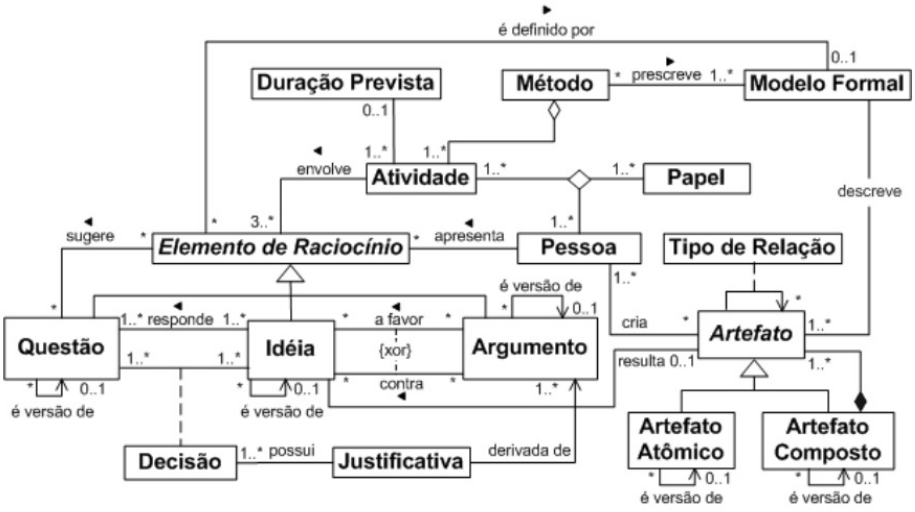

Figura 2.1: Vocabulário da ontologia Kuaba [Medeiros 2006]

Nesta ontologia, os Elementos de Raciocínio representam os principais elementos da estrutura de *Design Rationale*. Dentre estes elementos, encontram-se: Questões, Ideias e Argumentos. O elemento Questão representa todos os problemas de design que são

levantados durante a modelagem de software. O elemento Ideia contém as propostas de soluções para os problemas representados pelo elemento Questão. O elemento Argumento contém os argumentos, contra e a favor, para cada ideia proposta.

O usuário pode apresentar diversas propostas de modelagem para um elemento pertencente ao domínio, ou seja, apresentar diversas Ideias que respondem à Questão de como modelar tal elemento. Porém nem todas as Ideias são aceitas. O elemento Decisão é responsável por representar se uma Ideia foi aceita ou rejeitada como solução para uma Questão. O elemento Justificativa registra as razões para uma Decisão que são derivadas dos argumentos apresentados para as ideias.

O elemento Artefato representa o resultado final do design e é descrito a partir do elemento Modelo Formal, que representa o metamodelo de design utilizado na produção do artefato. O elemento Atividade representa as atividades de modelagem de um sistema em si que são realizadas de acordo com um método de design. Para cada atividade são registrados os elementos de raciocínio utilizados por uma pessoa, que possui um papel específico na realização da atividade.

A Figura 2.2 mostra um exemplo de representação de *Design Rationale* utilizando os elementos de raciocínio do vocabulário da ontologia Kuaba durante o design de um modelo de classes da UML para o domínio de livraria.

Toda a representação do *Design Rationale* se inicia a partir de uma Questão inicial que define o problema principal. A partir deste ponto são geradas Ideias que respondem a essa Questão e, conforme descrito na ontologia, gera-se novas Questões para estas Ideias, Decisões e os demais elementos definidos.

No exemplo da Figura 2.2 foi utilizado o metamodelo da UML para gerar as questões e as ideias de design no registro de *Design Rationale*. De acordo com esse metamodelo, a principal Questão do design é: “Quais são os elementos do Modelo?”. O projetista pode propor ideias para responder a esta pergunta, que correspondem aos elementos pertencentes ao domínio que se está modelando como, por exemplo, “Livro”, “Autor” e “Editora”. A partir da definição dessas ideias, geram-se novas questões específicas sobre como modelar cada conceito do domínio. Esta questão também será endereçada por outras ideias. Porém, dentro do contexto do metamodelo UML, as ideias que endereçam a questão raiz são chamadas

Ideias de Domínio. Todas as Ideias que endereçam Questões que não são a questão raiz são chamadas de Ideias de Design. As Ideias de Design representam as possíveis opções de design fornecidas pelo metamodelo UML que foram selecionadas para modelar uma Ideia de Domínio. Em um diagrama de classes da UML, por exemplo, as possíveis Ideias de Design que podem ser utilizadas para representar uma Ideia de Domínio são: Classe, Atributo, Operação, Associação, Interface, etc. No exemplo também são ilustradas algumas decisões tomadas pelo projetista. Essas decisões são indicadas com os rótulos "A" para ideias aceitas e "R" para as ideias rejeitadas.

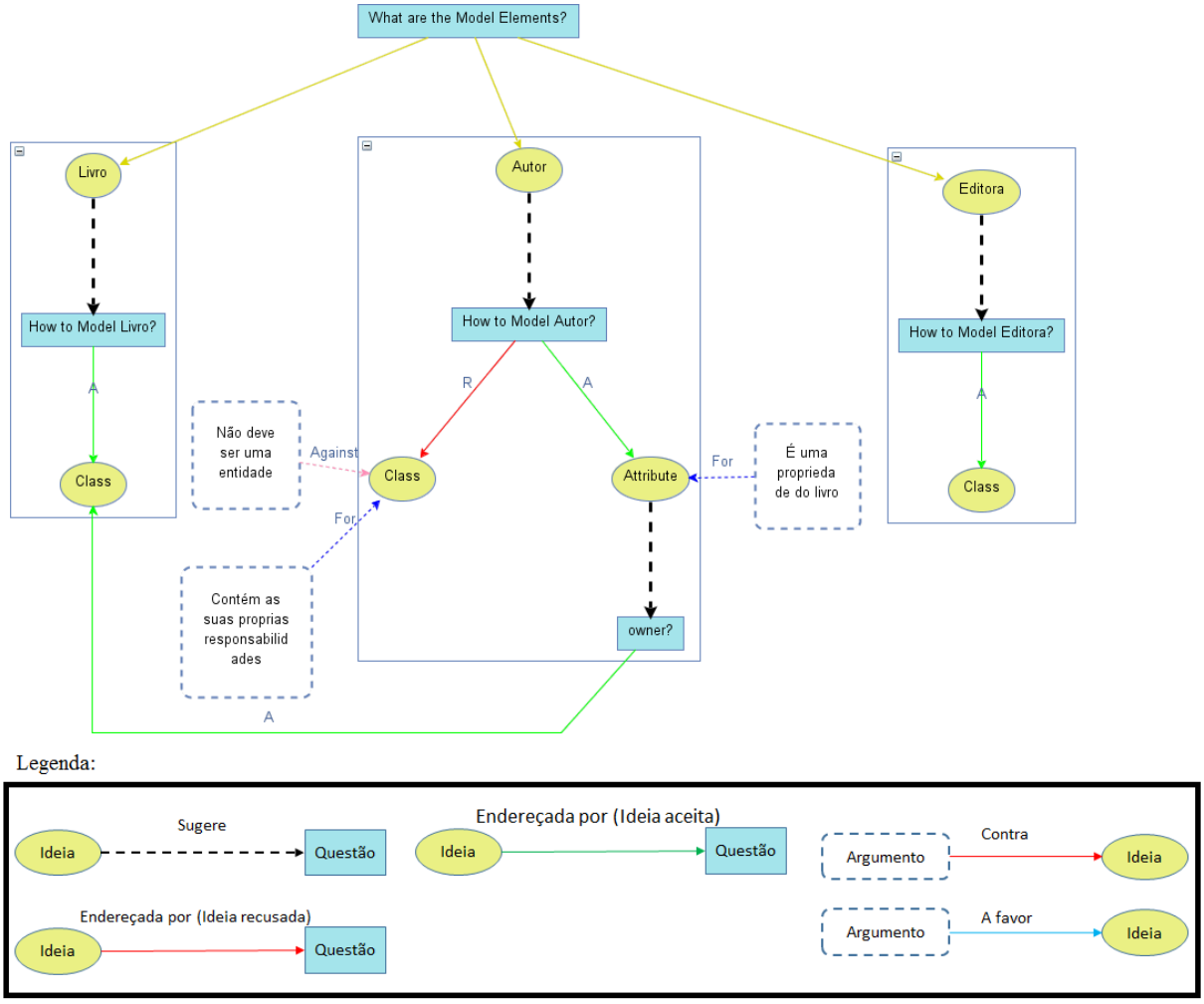

Figura 2.2: Grafo de *Design Rationale* para um domínio de livraria, usando a ontologia Kuaba.

2.2.1 A Linguagem OWL

A ontologia Kuaba é descrita pela linguagem OWL (*Web Ontology Language)* [OWL 2004], uma linguagem criada especificamente para a definição, manipulação e processamento

de ontologias e a representação de conhecimento. Esta linguagem segue as premissas da Web Semântica, ou seja, todo o conteúdo gerado pela mesma, além de ser compreensível para um leitor humano, também pode ser processado por um computador, para a realização de inferências a partir do conteúdo lido.

A OWL oferece três sublinguagens distintas, cada uma focada em um tipo diferente de comunidade e usuário. A *OWL Lite* é focada em usuários que precisam somente de uma classificação hierárquica e restrições simples (esta sublinguagem só aceita cardinalidades de 0 e 1, por exemplo). A *OWL DL* possui foco na área de lógica descritiva, pois fornece mais recursos para expressividade das ontologias. Já a *OWL Full* possui todos os recursos disponíveis pela OWL, oferecendo mais liberdade semântica e sintática. A ontologia Kuaba utiliza a sublinguagem OWL DL em sua especificação.

Além das sublinguagens, a OWL também contém diversas estruturas sintáticas distintas, permitindo uma maior flexibilidade na declaração de elementos pertencentes a uma ontologia. A sintaxe utilizada pela Kuaba é a RDF/XML [RDF 2004], onde as estruturas dos elementos seguem a arquitetura XML, utilizando *tags* que contém declarações sintáticas de elementos RDF. Tais elementos são considerados metadados, ou seja, descrições de elementos que são pertencentes à ontologia.

Os principais elementos dentro da estrutura OWL são *Classes*, *Object Properties* e *Data Properties*, *Object Property Domain*, *Data Property Domain*, *Data Property Ranges*, *Declarations* e *SubclassOf*. Classes representam estruturas abstratas que podem ser elementos do mundo real ou apenas a representação de ideias, por exemplo. Similar à UML, a classes podem possuir diversos atributos e relações com outras classes. Sua representação segue o padrão: <**Class IRI=**”Nome_da_Classe”/>, onde a sintaxe **Class** é a declaração de que a *tag* define uma classe e a sintaxe **IRI=** é sucedida pelo nome da classe que é declarada.

O elemento *Object Property* define uma relação entre classes existentes dentro da ontologia. Este elemento permite representar as associações existentes, criando uma maior flexibilidade ao se gerar os elementos da linguagem. Sua representação na KSE tem o formato: <**ObjectProperty IRI=**"#”Nome_da_Relacao"/>. A sintaxe **ObjectProperty** é a definição da representação da *tag* e a sintaxe **IRI=** é sucedida pelo nome da relação.

Data types definem elementos diretamente associados a classes, porém cuja representação não é complexa o suficiente para a criação de classes para representá-las. Entre os tipos de dados disponíveis estão: *string*, *boolean*, *integer*, *decimal*, *float*, entre outros. São utilizadas duas *tags* para realizar sua representação. A primeira define o nome do tipo de dado: <**DataProperty IRI=**#Nome_da_propriedade"/>. A segunda define o tipo de dado que este elemento possui: <**Datatype abbreviatedIRI**="xsd:dateTime"/>, onde a sintaxe **abbreviatedIRI=** é sucedida por xsd: "Tipo_de_Dado".

A definição da relação de posse entre uma classe e uma *Data Property* é realizada pela *DataPropertyDomain*, que contém a declaração de uma *Data Property*, seguida pela lista de Classes que possuem esta propriedade, sua declaração segue a forma:

```
<DataPropertyDomain>
   <DataProperty IRI="Nome_da_propriedade"/>
   <Class IRI="Nome_da_Classe"/>
        ...
</DataPropertyDomain>
```

De maneira análoga, a sintaxe *ObjectPropertyDomain* define a relação de posse entre um *Object Property* e uma classe:

```
<ObjectPropertyDomain>
   <ObjectProperty IRI="Nome_da_propriedade"/>
   <Class IRI="Nome_da_Classe"/>
        ...
</ObjectPropertyDomain>
```

A sintaxe *DataPropertyRange* é utilizada para definir um tipo de dado de um *DataProperty*. A OWL possui uma série de tipos de dados distintos, como inteiros, strings, booleanos, hexadecimais, entre outros. Sua declaração segue o formato:

```
<DataPropertyRange>
   <DataProperty IRI="#NOME_DA_DATA_PROPERTY"/>

   <Datatype abbreviatedIRI="xsd:TIPO_DE_DADI"/>
</DataPropertyRange>
```

As sintaxes *Declaration* contém as declarações dos elementos pertencentes à ontologia descrita. Portanto contém a declaração de todas as *Classes*, *Data Properties* e *Object Properties*, como no exemplo abaixo.

```
<Declaration>
   <Class IRI="#NOME_DA_CLASSE"/>
</Declaration>
```

Por fim, a sintaxe *SubClassOf* contém a organização hierárquica entre classes da ontologia. Também de maneira similar à UML, pode-se definir *SuperClasses* e *Classes* as quais descendem dessas *SuperClasses*. Dentro da declaração deste elemento, define-se: a superclasse, a cardinalidade mínima desta relação, o nome desta relação e o nome da subclasse desta relação, como no exemplo a seguir.

```
<SubClassOf>
   <Class IRI="#NOME_DA_SUPERCLASSE"/>
      <ObjectMinCardinality cardinality="1">
         <ObjectProperty IRI="#NOME_DA_RELACAO"/>
         <Class IRI="#NOME_DA_SUBCLASSE"/>
      </ObjectMinCardinality>
</SubClassOf>
```

2.3 Reengenharia de Software

Software desenvolvido com qualidade deve prover as funcionalidades especificadas e ser seguro. Além disto, deve ser manutenível, confiável e utilizável [Sommerville 2008]. Porém o software surge como resultado de um esforço aplicado em determinado período de tempo. À medida que este tempo passa, podem ser detectados erros, *bugs*, ou até mesmo ausência de compatibilidade entre os requisitos levantados para o software e os requisitos atendidos. Nestes casos, assim como em outros, pode ser necessária a aplicação de um processo de reengenharia de software.

De acordo com [Sommerville 2008], reengenharia de software pode ser realizada com o intuito de melhorar a estrutura e a compreensão de softwares legados. A reengenharia pode envolver redocumentação de sistema, refatoração de arquitetura, atualização e modificação de estrutura e os valores de dados do sistema. Porém a funcionalidade do software não deve ser

modificada e mudanças mais complexas em sua arquitetura devem ser evitadas. Entre as principais atividades previstas no processo de reengenharia de software, encontram-se: tradução do código-fonte, engenharia reversa, documentação do software, melhorias na estrutura do software, modularização do software e reengenharia dos dados.

Existem diversos sintomas específicos que podem ser exibidos por um software e que podem representar a necessidade da realização de um processo de reengenharia. Entre eles, encontram-se: ausência de documentação, desenvolvedores do sistema que não estão mais presentes, muito tempo para realização de mudanças simples, entendimento limitado do sistema, entre outros. Além disso, os problemas técnicos que motivam a reengenharia costumam ser bem similares: forte acoplamento entre os componentes da aplicação, camadas de software mal distribuídas, mal-uso de herança, entre outros fatores.

Baseado nestas necessidades, os autores em [Demeyer & Ducasse & Nierstraz 2008] definem padrões de projeto que podem ser utilizados na aplicação de reengenharia de software. Entre os quais encontram-se: *Read all the Code in One Hour*, uma técnica utilizada para ambientar o engenheiro de software ao código-fonte do software. *Skim the Documentation*, utilizada pelo engenheiro para verificar informações importantes que podem ser encontradas na documentação do software. *Split up God Classes*, técnica utilizada para solucionar problemas de acoplamento na estrutura do código do software.

Estes padrões são utilizados em distintas etapas no processo de reengenharia, da análise do código e da estrutura do software até a realização das alterações no código-fonte. Apesar de a KSE ainda não ser considerada um software legado, o uso das técnicas descritas neste trabalho é importante na melhoria e adequação da ferramenta a alguns requisitos, descritos no decorrer deste trabalho.

Capítulo 3

A Ferramenta KSE (*Kuaba Software Engineering*)

A KSE é uma ferramenta de modelagem de softwares orientados a objetos que utiliza a abordagem Kuaba para realizar a captura semi-automática de *Design Rationale*. A ferramenta tem como base a ArgoUML [ArgoUML 2008], um software open-source desenvolvido em Java. Toda a estrutura da ArgoUML é focada na modelagem de software seguindo o metamodelo da UML. A captura e representação de *Design Rationale* é de responsabilidade do subsistema Kuaba, que foi integrado à estrutura da ArgoUML reusada na KSE.

O subsistema Kuaba [Nunes 2010] é um módulo isolado projetado para ser independente da ferramenta e do metamodelo de design utilizados, minimizando o impacto das mudanças necessárias no caso de uma implantação em outras ferramentas CASE. A estrutura do subsistema Kuaba foi definida de modo que o subsistema se responsabilize por observar e processar as ações realizadas durante a produção de um diagrama para gerar o *Design Rationale*.

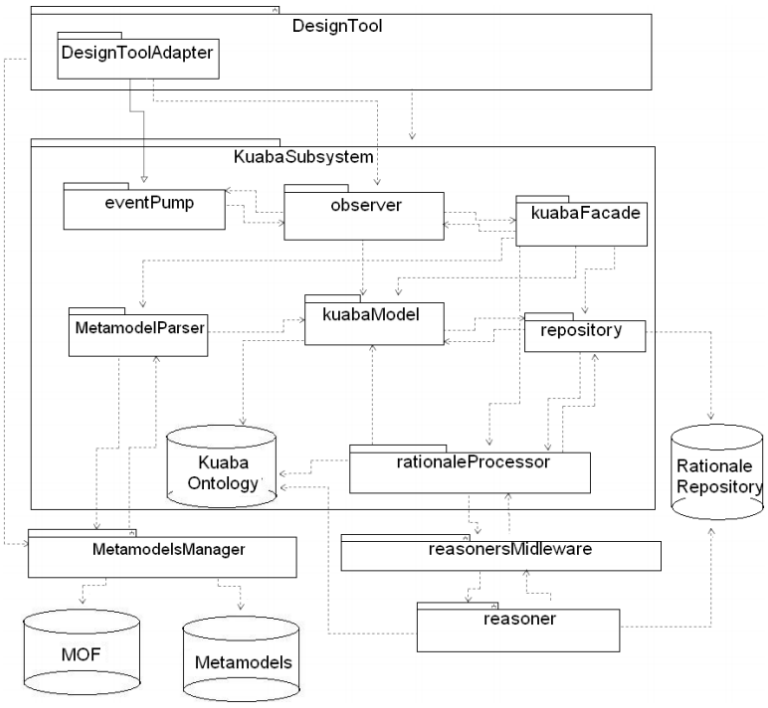

Figura 3.1: Arquitetura do subsistema Kuaba [Nunes & Medeiros 2009]

A Figura 3.1 exibe a arquitetura do subsistema Kuaba. O pacote *DesignToolAdapter* contém as classes que são o elo de ligação entre o subsistema e a ferramenta de design. As mudanças realizadas no modelo devem ser captadas por classes dentro deste pacote e repassadas ao subsistema de modo que este possa processá-las.

KuabaSubsystem contém toda a estrutura lógica de processamento da abordagem Kuaba. O pacote *eventPump* é responsável por notificar as mudanças detectadas para as respectivas classes do pacote *observer*. Esse pacote, que contém classes que implementam o padrão *Observer* [Gamma et al 1995], captura os eventos referentes às ações realizadas pelo usuário no diagrama UML e gera o grafo de *Design Rationale* de acordo com a abordagem Kuaba utilizando os elementos correspondentes do metamodelo da UML. Para realizar as modificações necessárias nesse grafo de *Design Rationale*, este pacote notifica objetos do pacote *kuabaModel* que realiza a manipulação direta dos elementos definidos na ontologia Kuaba. Como a manipulação do grafo de *Design Rationale* pode ser uma tarefa complexa, o *kuabaFacade* também pode ser utilizado para auxiliar nestas manipulações. Este pacote realiza a implementação do padrão *facade* [Gamma et al 1995].

A Figura 3.2 detalha a relação entre os pacotes *Observer*, *eventPump* e *Facade*. É possível verificar no pacote *eventPump* que, ao receber alguma notificação de mudança no diagrama, seja pela criação, alteração ou remoção de algum elemento, a classe *ArgoUMLEventPumpAdapter* notifica a classe *ModelElementObserver*, no pacote *Observer*. Esta classe processa a mudança realizada no modelo com o auxílio da classe *KuabaFacade* no pacote *Facade*.

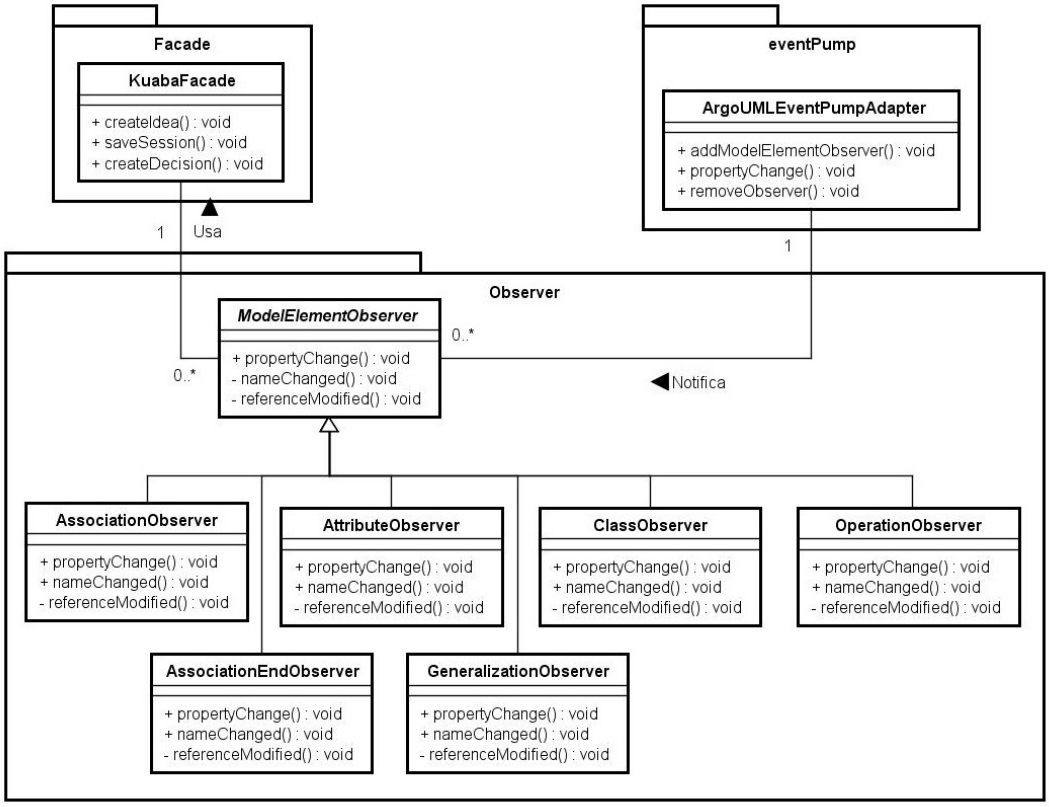

Figura 3.2: Detalhamento do pacote *Observer* e suas principais relações

O pacote *repository* é responsável por manipular os repositórios de *Design Rationale*. Este pacote contém classes que manipulam os elementos da ontologia conforme suas relações descritas em OWL. O pacote *rationaleProcessor* provê classes que permitem à KSE fazer o reúso de design de software, permitindo ao projetista iniciar o design de um artefato a partir de outro artefato já existente e que pertence ao mesmo domínio, conforme descrito em [Diniz 2013]. Por fim, o *metamodelParser* contém as classes que interpretam o metamodelo que é utilizado pela ferramenta principal. Na KSE, o metamodelo analisado é o metamodelo da UML.

A Figura 3.3 ilustra a representação dos elementos da ontologia no subsistema. A criação dos elementos descritos na ontologia Kuaba é realizada seguindo o padrão *factory* [Gamma et al. 1995]. Na KSE, a manipulação destes elementos é realizada através de interfaces. Os objetos da ontologia não são diretamente manipulados. Esta abordagem fornece uma maior solidez na manipulação dos elementos.

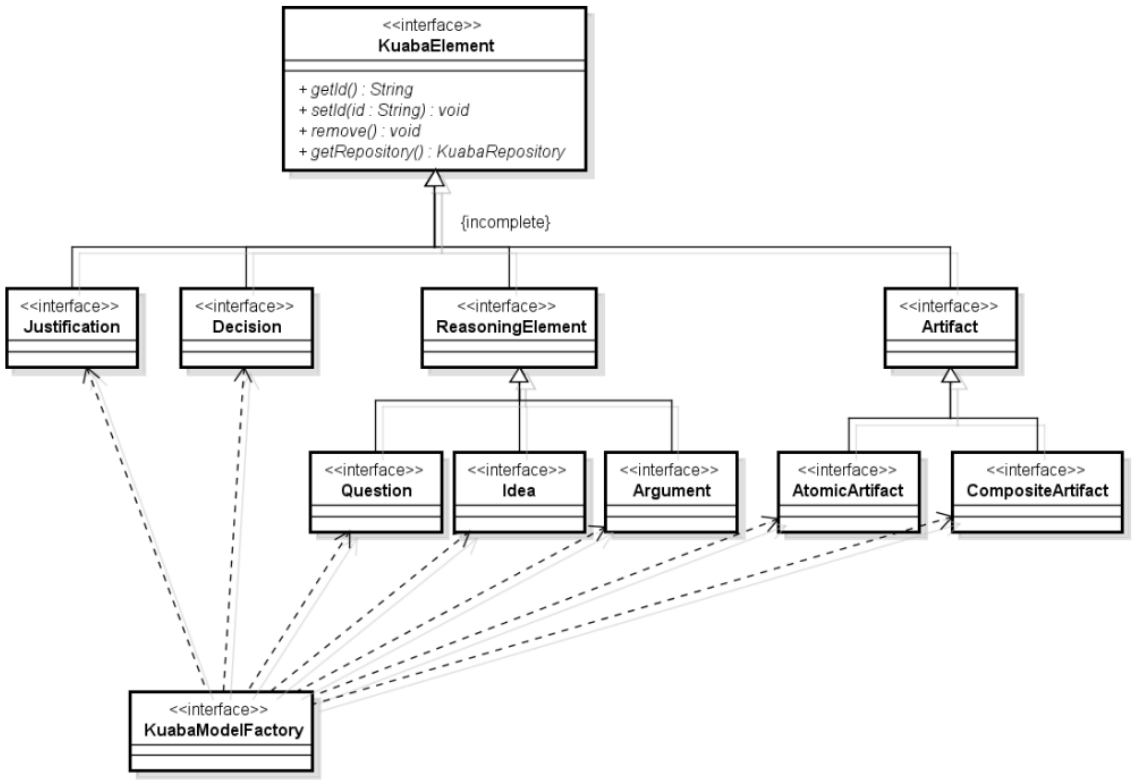

Figura: 3.3: O padrão *Factory* utilizado na instanciação dos elementos da ontologia Kuaba [Diniz 2013]

Como dito anteriormente, a KSE realiza a captura e representação de *Design Rationale* de modo semiautomático. A KSE captura as mudanças realizadas no artefato para gerar as ideais de domínio e as ideias de design. Em seguida, baseada no metamodelo, a ferramenta gera as questões levantadas por estas ideias, quando conveniente. Quando estas ideias são geradas, modificadas ou removidas a ferramenta captura argumentos para estas mudanças, conforme exibido na Figura 3.4.

A janela em questão é criada para capturar o argumento a favor da utilização desta ideia de design. Porém são geradas janelas também para captura de argumentos contra, seja quando se modifica o nome de uma classe, ou quando se remove uma do modelo. Este comportamento é replicado para atributos, operações, associações e generalizações no diagrama.

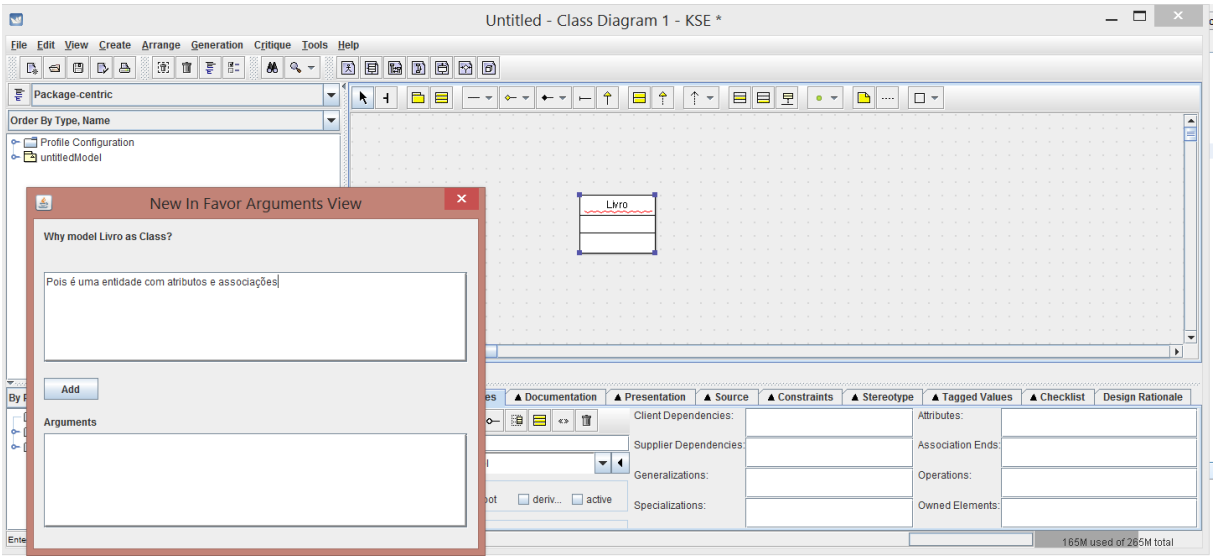

Figura 3.4: Janela exibida quando se cria uma nova classe em um diagrama de classes UML

A KSE também permite a edição das ideias e dos argumentos através de uma aba, localizada na seção inferior da janela principal do software.

É possível fazer uma visualização do grafo de *Design Rationale* que é gerado pela KSE, como ilustra a Figura 3.5. A visualização pode ser tanto vertical como horizontal. Além disso, são exibidas Ideias, Questões e Decisões geradas para o modelo atual. A ferramenta nem sempre exibe todas as ideias de design associadas a uma ideia de domínio, pois o gráfico pode acabar se tornando muito extenso e de difícil compreensão para o usuário.

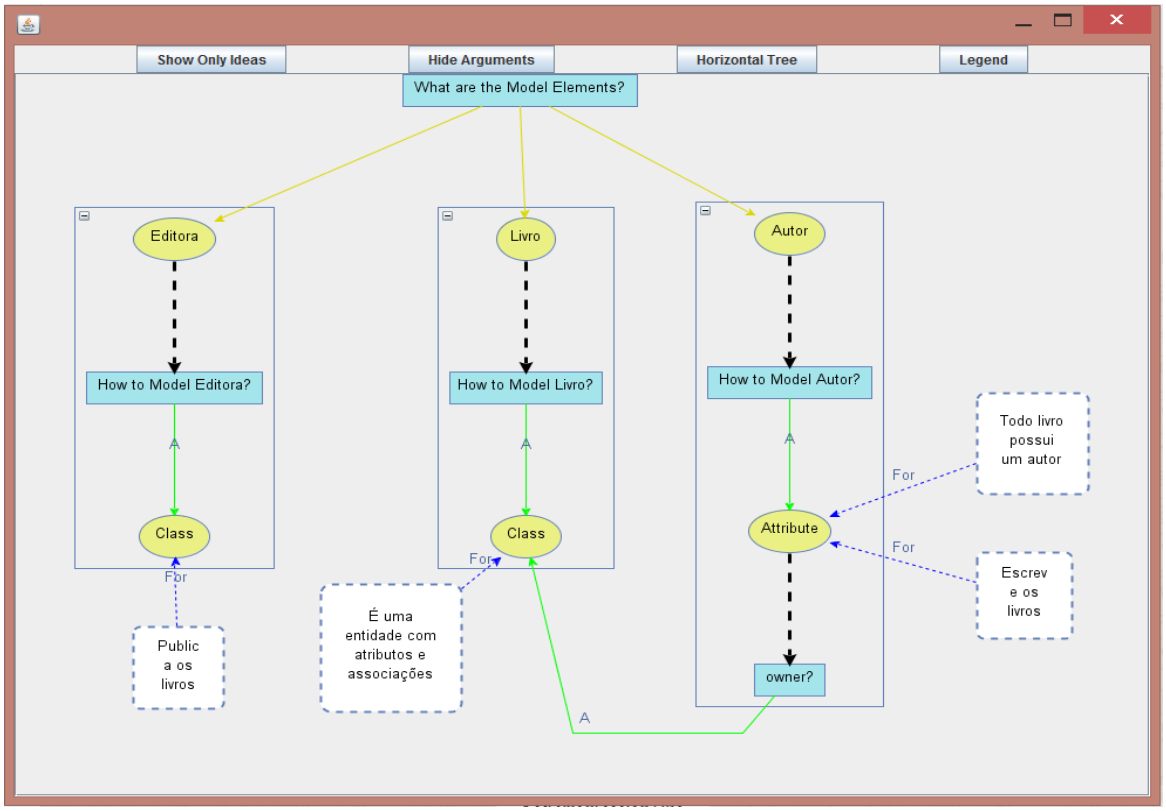

Figura 3.5: Grafo do *Design Rationale* gerado pela KSE

Na KSE, o registro automático de *Design Rationale* não se limita somente às questões e ideias. Como mostra a Figura 3.5, as decisões (setas rotuladas com “A”) também são geradas pela ferramenta para as ideias de design propostas, em tempo de execução. A KSE foi projetada com o intuito de gerar e manipular as decisões em tempo real, ou seja, indicar se ideias foram aceitas ou recusadas enquanto o projetista realiza a modelagem. Porém, esta manipulação não ocorre de maneira adequada na última versão da ferramenta. Ao modificar as decisões para ideias de design, a KSE não atualiza estas decisões para suas subárvores. Ideias que deveriam ser exibidas como recusadas aparecem como aceitas. Este comportamento é descrito em mais detalhes no capítulo seguinte.

Capítulo 4

Reengenharia de software aplicada na KSE

Como visto, a KSE é capaz de detectar as ações realizadas em um diagrama durante a modelagem e gerar o registro de *Design Rationale*, utilizando os elementos do metamodelo da UML. Porém, a captura e representação de alguns elementos de *Design Rationale* ainda não estão sendo feitas de forma adequada.

4.1 Problemas encontrados

Na KSE, as decisões relacionadas às ideias de design são registradas em tempo real durante a própria modelagem. Uma das propostas da ferramenta é que o projetista possa visualizar, enquanto modela, o grafo de *Design Rationale* gerado para o diagrama. Sendo assim, a ferramenta deve indicar tanto decisões aceitas quanto decisões rejeitadas até este momento. No entanto, a KSE não atualiza as decisões no grafo de *Design Rationale* quando modificações no diagrama alteram as decisões de aceitas para recusadas, gerando inconsistências.

A Figura 4.1 reproduz o cenário descrito no parágrafo anterior para um caso onde duas classes, Livro e Editora, estão relacionadas pela associação Publica. Ao se remover esta associação, a subárvore de *Design Rationale* para a ideia de domínio Publica não foi atualizada com decisões recusadas.

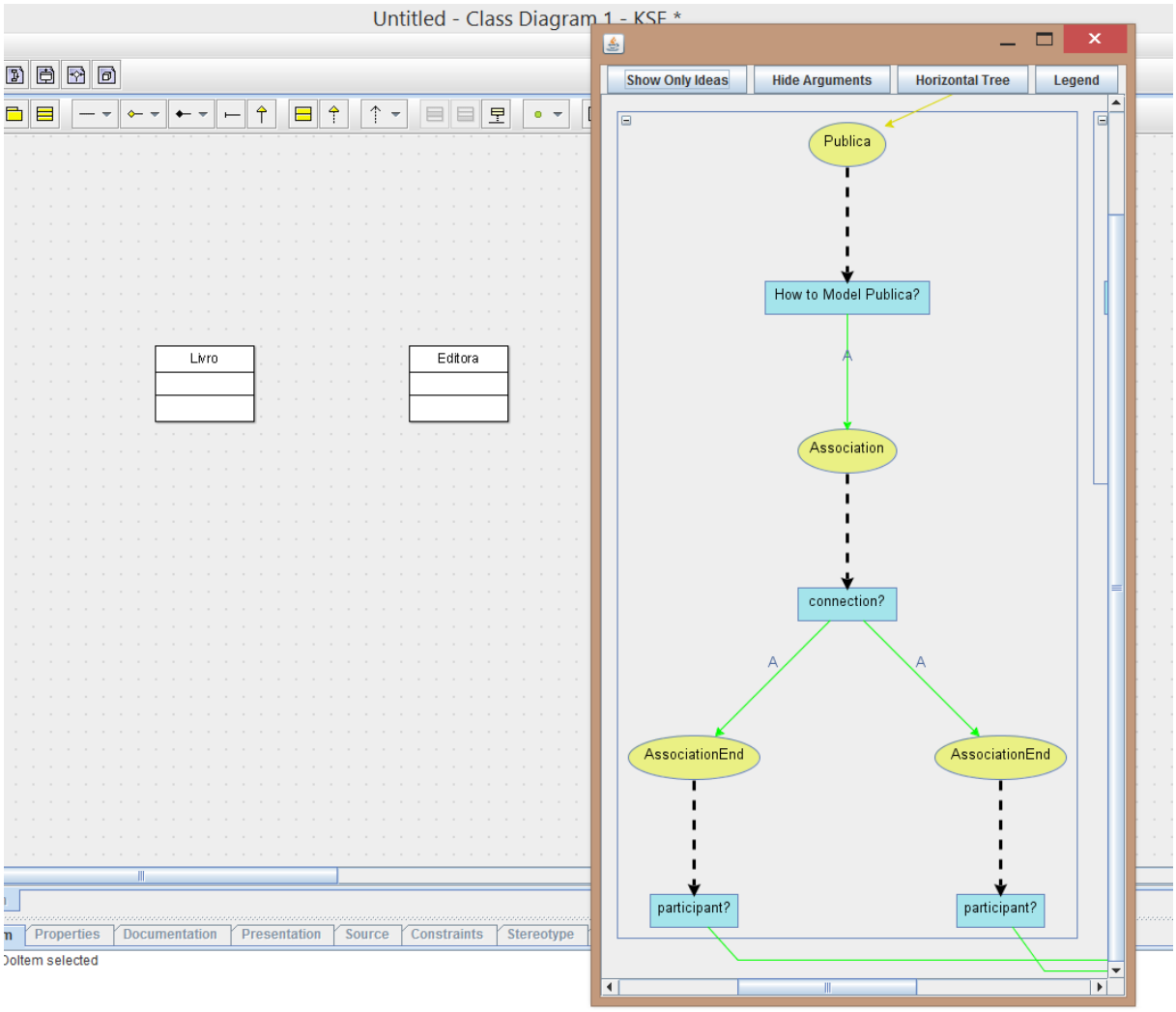

Figura 4.1: Exemplo de inconsistências na geração do grafo de *Design Rationale*

Outro caso de inconformidade entre a KSE e a abordagem Kuaba está na manipulação das ideias de domínio. Ao remover uma ideia de domínio do diagrama e inseri-la novamente com outra ideia de design associada, a ideia de domínio é duplicada no modelo de *Design Rationale*. Além disto, a ideia de design original também não é atualizada como recusada.

A Figura 4.2 ilustra este erro, onde a ideia de domínio Autor é duplicada e cada uma contém uma ideia de design distinta associada, sendo ambas consideradas aceitas.

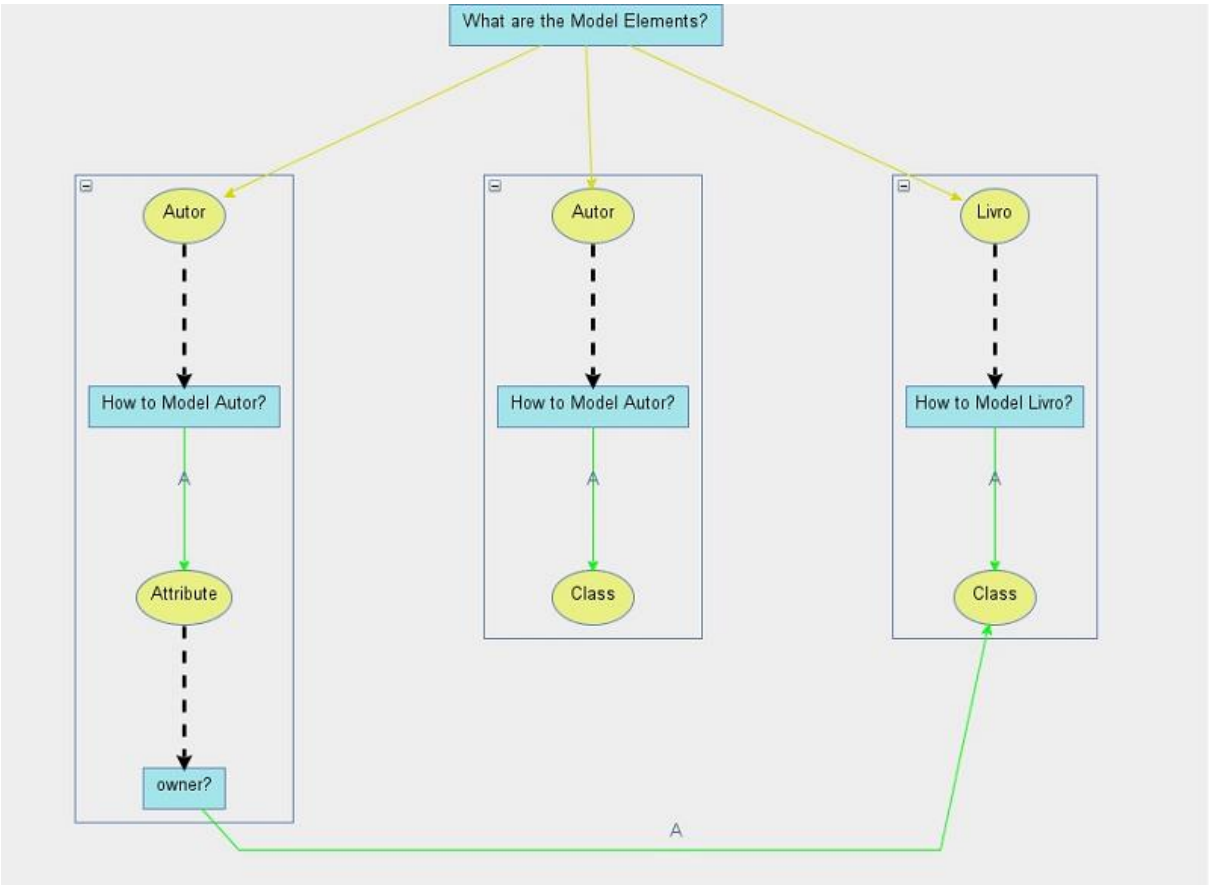

Figura 4.2: Ideia de domínio repetida

4.2 Técnicas de Reengenharia utilizadas

Devido aos problemas descritos na seção anterior, foi considerada a necessidade de aplicação de reengenharia de software na KSE para adequá-la à ontologia Kuaba. Para a realização da reengenharia, foram aplicados os padrões descritos em [Demeyer & Ducasse & Nierstraz 2008], reproduzidos nos parágrafos seguintes.

Realizar mudanças não triviais em softwares complexos não é uma tarefa simples para o desenvolvedor, ainda mais quando se tem conhecimento limitado do código-fonte de uma ferramenta. Os primeiros passos para interagir com o código-fonte devem servir como uma introdução geral ao sistema. Para suprir esta necessidade, existem duas técnicas que podem ser aplicadas. Estas são:

1. Leitura de todo o código fonte em uma hora. Esta técnica visa verificar o estado do software pela realização de uma revisão de código intensa. Com esta técnica, é possível perceber padrões de desenvolvimento, classes acopladas, tipos de códigos que podem indicar problemas (*Code Smells*), entre outros.

2. Leitura de documentação. A documentação de uma ferramenta pode servir como um forte apoio ao desenvolvedor na compreensão de uma ferramenta, ao indicar estruturas de código utilizadas. Além disso, pode indicar requisitos e funcionalidades que são implementadas na ferramenta.

Ambas as técnicas foram aplicadas na KSE como forma de entendimento inicial da sua estrutura e lógica. Após aplicar a primeira técnica, Leitura do código fonte em uma hora, foi possível compreender de maneira mais detalhada a relação entre a ArgoUML e o subsistema Kuaba, o que gerou as seguintes observações:

- O pacote *eventPump* do subsistema Kuaba encontra-se implementado também parcialmente na ferramenta ArgoUML.
- Os elementos da ontologia Kuaba que são declarados em OWL são convertidos para interfaces e classes Java para que possam ser processados. Estão localizados no pacote *kuabaModel*.
- Para gerar os arquivos com a descrição de *Design Rationale* específicos, utiliza-se uma API que valida os elementos processados no subsistema Kuaba com a descrição da ontologia em OWL.

Para aplicação da segunda técnica, Leitura da Documentação, foi utilizado como fonte de documentação o relatório técnico do desenvolvimento inicial da ferramenta [Nunes, 2010]. O relatório contém uma descrição útil da estrutura geral do sistema, além de detalhar os padrões de projeto utilizados para instanciar e manipular elementos da ontologia Kuaba, através das classes *kuabaModelFactory* e *kuabaFacade*. Outro fator importante a se ressaltar sobre o documento é sua descrição quanto à implementação do padrão *Observer* como meio de interação entre o subsistema Kuaba e a ArgoUML.

Com base nas informações extraídas, foi levantada a hipótese de que os problemas no processamento das decisões e a duplicidade das ideias de domínio poderiam ter relação com as classes referentes aos padrões *observer* ou *facade*. Portanto, em seguida, foi realizada uma análise do código-fonte, com enfoque nas classes que fazem parte destes padrões. Após esta análise, foram detectados diversos problemas que, entre outros, também acarretavam os erros apresentados no início deste capítulo. Estes foram:

- Classe *kuabaHelper*, que contém diversas funções distintas, que não contém contexto semântico dentro da arquitetura da aplicação.
- O filtro para tratamento de eventos disparado pela ferramenta ArgoUML impede que o subsistema Kuaba trate adequadamente relações de generalização e realização de interfaces.
- Má implementação do padrão *Observer*. Os eventos notificados para a KSE são recebidos e tratados por diversas subclasses que apenas delegam para a mesma superclasse a responsabilidade de realizar a modificação no *Design Rationale*.
- O subsistema provê somente um algoritmo para modificação de decisão no *Design Rationale* gerado pela ferramenta. Este algoritmo não realiza a modificação da decisão na subárvore de ideias.
- Não são criados novos elementos de *Design Rationale* quando se modifica o nome de uma associação no modelo UML.
- Uma nova ideia de domínio é instanciada sempre que se criar um novo elemento no diagrama da ArgoUML.

4.3 Analise das mudanças necessárias

Identificados os problemas principais, foi preciso analisar quais mudanças, dentro do escopo destes elementos, seriam necessárias para que a ferramenta pudesse atender de maneira mais satisfatória a manipulação dos elementos da ontologia Kuaba. Com base nos problemas levantados, decidiu-se realizar correções no tratamento dos eventos recebidos pelo subsistema Kuaba, ou seja, na implementação das classes pertencentes ao padrão *observer*.

Segundo [Demeyer & Ducasse & Nierstraz 2008], classes que assumem muitas responsabilidades, também conhecidas como *god classes*, apresentam um grande desafio para a evolução da aplicação. Além disto, é difícil identificar onde mudanças realizadas nestas classes afetariam o sistema como um todo. Como solução para este problema, sugere-se que as responsabilidades destas classes sejam redistribuídas para classes associadas.

Dentro do contexto da KSE, a classe *modelElementObserver* pode ser considerada como *god class*, por ser responsável pela realização do tratamento das mudanças realizadas em todo o modelo. As subclasses que recebem as notificações de mudanças delegam para esta superclasse esta responsabilidade. Portanto, o plano de ação elaborado para realizar correções

na KSE consistiu na alteração da implementação das classes que integravam o padrão *observer*. O objetivo desta correção foi delegar às subclasses de *modelElementObserver* a responsabilidade das mudanças no *Design Rationale* para os respectivos elementos do diagrama que estas subclasses observavam.

Os eventos capturados pelo subsistema Kuaba e enviados para as respectivas classes no padrão *Observer* podem ser organizados em três categorias distintas:

1. Mudança do nome de um elemento do diagrama.
2. Remoção do nome de um elemento do diagrama.
3. Mudança da referência de um elemento do diagrama.

Seja, por exemplo, a modelagem de um atributo chamado “Nome”. Após o projetista nomear este atributo, no subsistema Kuaba a classe *attributeObserver* recebe uma notificação de mudança no diagrama. Esta notificação contém: o tipo de mudança realizada (que se encaixa em uma das três categorias listadas acima), o nome do atributo e uma referência a este elemento. Com estas informações, a *attributeObserver* requisita a criação de uma nova ideia de domínio, contendo o nome do atributo criado e uma ideia de design *Attribute* que responde à questão sugerida por esta ideia de domínio (Como Modelar Nome?). Por fim, também é gerada uma Decisão, com o status Aceita, para a ideia de domínio criada “Nome” e outra Decisão, também com o status Aceita, para a ideia de design “*Attribute*”. Os quadros 4.1 e 4.2 mostram os trechos de código correspondentes a esse exemplo.

```
public void propertyChange(PropertyChangeEvent evt) {
   super.propertyChange(evt);
}
```

Código 4.1: Método *propertyChange* na classe *AttributeObserver*

O comando super.propertyChange(evt) faz com que a classe ModelElementObserver seja notificada, para realizar o tratamento do evento ‘evt’.

```
public void propertyChange(PropertyChangeEvent evt) {
   RefObject source = (RefObject)evt.getSource();
   if(evt.getPropertyName().equals("remove")){
         KuabaSubsystem.eventPump.removeObservers(source);
         Idea domainIdea =
         KuabaSubsystem.facade.getIdeaOnSession(KuabaSubsystem.resolver.resolveXmiId(source));
         KuabaSubsystem.facade.domainIdeaSubTreeDecision(domainIdea, false);
         Question howModelQuestion = (Question)domainIdea.listSuggests().next();
         List<Idea> acceptedDesignIdeas =
KuabaHelper.getAcceptedAddressedIdeas(howModelQuestion);
         for (Idea idea : acceptedDesignIdeas) {
            ArgumentController controller =new ObjectsToArgumentController(null,new
Idea[]{idea},howModelQuestion);
             controller.render();
         }
         KuabaSubsystem.facade.domainIdeaDesconsidered(domainIdea);
         return;
   }
   MofClass metaClass = (MofClass)source.refMetaObject();
   String designIdeaText = (String)metaClass.refGetValue("name");
   if(evt.getPropertyName().equals("name")){
      String newName = "";
      String oldName = "";
      if(evt.getNewValue() != null)
         newName = (String)evt.getNewValue();
      if(evt.getOldValue() != null)
          oldName = (String)evt.getOldValue();
      this.nameChanged(newName, oldName,
designIdeaText,"_"+KuabaSubsystem.resolver.resolveXmiId(source));
   }
   if(evt.getNewValue() instanceof RefObject)
      this.referenceModified((RefObject)evt.getNewValue(), evt);
}
```

Código 4.2: Método propertyChange na classe ModelElementObserver

No código 4.2 é possível observar que a classe *attributeObserver* delega à classe *ModelElementObserver* a responsabilidade de tratar estes três tipos de eventos. Esta transferência de responsabilidade é o principal fator pelo qual a manipulação do *Design*

Rationale se encontra tão suscetível a erros. Portanto, as melhorias planejadas para as subclasses devem levar em conta o tratamento destes três tipos de entradas. Como resultado do planejamento da solução, foram elaborados três métodos distintos que são responsáveis por realizar estas tarefas: *nameChanged*, *removedElement* e *referenceModified*, onde cada um corresponde a uma das três categorias listadas acima, respectivamente.

O código 4.3 descreve o uso destes métodos na classe *classObserver*. Ao receber uma notificação de mudança no diagrama, o método *propertyChange* recebe um evento que indica em qual das três categorias descritas anteriormente esta mudança se encaixa e a referência para a classe que foi modificada e o nome da mesma.

```
public void propertyChange(PropertyChangeEvent evt) {
   RefObject source = (RefObject)evt.getSource();
   if(evt.getPropertyName().equals("remove"))
      this.removedElement(source);
   if(evt.getPropertyName().equals("name"))
   {
       (MofClass)source.refMetaObject();
       String designIdeaText = (String)metaClass.refGetValue("name");
       if(!evt.getNewValue().equals(""))
           this.nameChanged((String)evt.getNewValue(), (String)evt.getOldValue(),designIdeaText, "_"+
KuabaSubsystem.resolver.resolveXmiId(source));
   }
   if (evt.getNewValue() instanceof RefObject)
      this.referenceModified((RefObject)evt.getNewValue(), evt);
}
```

Código 4.3: Método *propertyChange* da classe *classObserver*

4.4 Refatoração de código

A primeira alteração realizada na ferramenta KSE corresponde à classe *classObserver*. A melhoria realizada buscou manter a responsabilidade original da classe, ou seja, obter as mudanças realizadas em classes declaradas no diagrama e processar o grafo de *Design Rationale* para estas respectivas classes, apenas corrigindo os erros de manipulação das decisões e na duplicidade das ideias descritos anteriormente.

Na classe *classObserver*, os métodos definidos apresentam os seguintes pseudo-códigos:

```
1 .  nameChanged(novoNome, nomeAntigo, ideia de design, ID da classe)
se for o primeiro nome dado a esta classe
        KuabaFacade -> criaNovaIdeiaDeDominio(novoNome, ideiadeDesign, IDdaclasse)
        ArgumentController -> ObtemArgumentoParaIdeiaAceita
Senão se já existe uma ideia de domínio para novoNome
        Ideia de Dominio = KuabaHelper -> ObtemIdeiaDeDominio(novoNome)
        KuabaFacade -> reconsideraIdeiaDeDominio(Ideia de Dominio)
        ArgumentController -> ObtemArgumentoParaIdeiaAceita
Senão se não existe uma ideia de domínio para novoNome
        KuabaFacade -> criaNovaIdeiaDeDominio(novoNome, ideiadeDesign, IDdaclasse)
        ArgumentController -> ObtemArgumentoParaIdeiaAceita
Se houver uma ideia de domínio para nomeAntigo
        KuabaFacade -> desconsideraIdeiaDeDominio(nomeAntigo)
        ArgumentController -> ObtemArgumentoParaIdeiaRecusada

2. removedElement(IDElemento)
        IdeiaDeDominio = KuabaFacade -> ObtemIdeiaDeDominio(IDElemento)
        KuabaFacade -> desconsideraIdeiadeDominio(IdeiaDeDominio)
        ArgumentController -> ObtemArgumentoParaIdeiaRecusada

3. referenceModified manteve o código da superclasse.
```

Pseudo-Código 4.1: Novo código em *classObserver*

Os pseudo-códigos descritos em 4.2 e 4.3 foram utilizados como base para a implementação das subclasses que tratam os outros elementos do metamodelo UML, porém respeitando as particularidades de cada elemento do metamodelo no *Design Rationale*. Na classe *associationObserver*, responsável por tratar as associações entre classes do diagrama de classes UML, foi necessário realizar a implementação de funções adicionais. Isto ocorre porque a subárvore de *Design Rationale* gerada para uma associação é composta de 6 níveis de pares Questão-Ideia, sendo estes, respectivamente: A ideia de domínio, a Questão sugerida pela ideia de domínio (*How to Model?*), a Ideia de design que responde esta questão (*Association*), a Questão sugerida por esta ideia (*Participant?*), duas Ideias de Design que respondem a esta questão, uma para cada extremidade da associação, a Questão sugerida para

cada uma destas duas ideias de design e as ideias de Design (classes) que endereçam estas Questões.

```
nameChanged(novoNome, nomeAntigo, ideia de design, ID da classe)
        Se for o primeiro nome dado a esta classe
                KuabaFacade -> criaNovaIdeiaDeDominio(novoNome, ideiadeDesign, IDdaclasse)
                ArgumentController -> ObtemArgumentoParaIdeiaAceita
Senão
        Se já existe uma ideia de domínio para novoNome
IdeiaDeDominio = KuabaHelper -> ObtemIdeiaDeDominio(novoNome)
                KuabaFacade -> reconsideraIdeiaDeDominio(Ideia de Dominio)
                ArgumentController -> ObtemArgumentoParaIdeiaAceita
        Senão
KuabaFacade -> criaNovaIdeiaDeDominio(novoNome, ideiadeDesign, IDdaclasse)
IdeiaDeDominio = KuabaHelper -> ObtemIdeiaDeDominio(novoNome)
    ArgumentController -> ObtemArgumentoParaIdeiaAceita
        Se houver uma ideia de domínio para nomeAntigo
                IdeiaRejeitada = KuabaFacade -> obterIdeiaDeDominio(nomeAntigo)
                KuabaFacade -> desconsideraIdeiaDeDominio(IdeiaRejeitada)
                copiaQuestao(IdeiaDeDominio,IdeiaRejeitada,2)
                ArgumentController -> ObtemArgumentoParaIdeiaRecusada
```

Pseudo-Código 4.2: Novo método *nameChanged* em *associationObserver*

```
removedElement(IDElemento)
        eventPump -> removeObserver(IDElemento)
        IdeiaDeDominio = KuabaFacade -> ObtemIdeiaDeDominio(IDElemento)
        KuabaFacade -> desconsideraIdeiadeDominio(IdeiaDeDominio)
        ArgumentController -> ObtemArgumentoParaIdeiaRecusada
        configuraDecisaoSubArvore(IdeiaDeDominio,recusada,2)
        referenceModified manteve o código da superclasse.
```

Pseudo-Código 4.3: Novo método *removedElement* em *associationObserver*

A função copiaQuestao() possui a responsabilidade de copiar as questões originalmente sugeridas pela Ideia rejeitada para a Ideia aceita. De acordo com a profundidade da subárvore que for fornecida, este algoritmo também utiliza a função copiaIdeia(), que realiza a mesma tarefa, porém para o elemento Ideia. Juntos, os dois

algoritmos são capazes de copiar toda a sub-árvore dos pares Questão-Ideia de uma Ideia de domínio específica.

```
copiaQuestao(IdeiaRejeitada,IdeiaAceita,profundidade)
  se profundade > 0
    se IdeiaAceita não sugere uma Questão
      profundidade = profundidade -1
       para cada Questao Q em IdeiaRejeitada faça
         Questao Q2 = KuabaFacade -> copiaQuestao(Q)
         IdeiaAceita -> sugereQuestao(Q2)
         copiaIdeias(Q,Q2,profundidade)
    senão
      profundidade = profundidade -1
      para cada Questao Q em IdeiaRejeitada faça
        para cada Questao Q2 em IdeiaAceita faça
          se Q == Q2
            copiaIdeias(Q,Q2,profundidade)
          senão
            Questao Q3 = Kuaba Facade -> copiaQuestao(Q)
            IdeiaAceita -> SugereQuestao(Q3)
            profundidade = profundidade – 1
            copiaIdeias(Q2, IdeiaAceita, profundidade
```

Pseudo-codigo 4.4: algoritmo que copia questões de uma subárvore

```
copiaIdeia(QuestaoRejeitada, QuestaoAceita, profunidade)
  se profundidade = 0
    se QuestaoRejeitada possui uma Decisão
      QuestaoRejeitada -> definirDecisão(false)
       Senão
         Ideia IdeiaRecusada = QuestaoRejeitada -> obterIdeia()
         KuabaFacade -> criarDecisão(QuestaoRejeitada, IdeiaRecusada, falso)
         Se QuestaoRejeitada possui alguma Ideia I que não endereça QuestaoAceita
           QuestãoAceita -> endereçadaPorIdeia(I)
         Se QuestaoAceita possui uma Decisao
           QuestaoAceita -> definirDecisão(true)
         Senão
           Ideia IdeiaAceita = QuestaoAceita-> obterIdeia()
           KuabaFacade -> criarDecisão(QuestaoRejeitada, IdeiaRecusada, falso)
```

```
Senão se profundidade > 1
  profundidade = profundidade -1
  Se QuestaoAceita não for endereçada por nenhuma Ideia
    Para cada Ideia I que endereça QuestãoRejeitada
    I -> definirDecisão(true)
    Ideia I2 = KuabaFacade -> copiaIdeia(I,QuestaoAceita)
    KuabaFacade -> criarDecisão(QuestaoAceita,I2,true)
    copiaQuestao(I,I2,profundidade)
  Senão
    Para cada Ideia I que endereça QuestaoRejeitada
      I -> configurarDecisão(false)
      Para cada Ideia I que endereça QuestaoAceita
        Se I não possui Decisão
          KuabaFacade -> criarDecisão(QuestaoAceita,I,true)
        Senão
          Decisão D = QuestaoAceita -> ObterDecisao(I)
          D -> definirDecisão(true)
          Para cada Ideia I que endereça QuestaoRejeitada
            ExisteIdeia = false
            Para cada Ideia I2 que endereça QuestaoAceita
              Se I == I2
                ExisteIdeia = true
                copiaQuestao(I,I2,profundidade)
                Se ExisteIdeia == false
                  Ideia I3 = KuabaFacade -> copiaIdeia(I,QuestaoAceita,true)
```

Pseudo-codigo 4.5: algoritmo que copia ideias de uma subárvore

A outra função, configuraDecisaoSubArvore() é executada quando se remove uma associação do diagrama. Este algoritmo percorre, nível a nível, a subárvore para modificar a Decisão entre os pares Questão-Ideia.

```
configuraDecisaoSubArvore(Ideia, profundidade)
  Se profundidade == 0
    Para cada Questao Q sugerida por Ideia
      Se Decisão D do par Q-Ideia é aceita
        D -> definirDecisão(false)
          ArgumentController -> ObtemArgumentoParaIdeiaRecusada
      Se profundidade > 0
        Profundidade = profundidade – 1
```

```
        Para cada Questão Q que é sugerida por Ideia
          Para cada Ideia I que endereça Q
            Se Existir uma Decisão D do par Q-I
              D -> definirDecisão(false)
            Senão
              KuabaFacade -> criarDecisão(Q,I,false)
              configuraDecisaoSubArvore(Q,I)
```

Pseudo-codigo 4.6: algoritmo que altera decisões de uma subárvore

As classes *attributeObserver* e *operationObserver*, que tratam modificações em atributos e operações, respectivamente, possuem implementações semelhantes.

```
nameChanged(novoNome, nomeAntigo, ideia de design, IDAtributo)
  Se for o primeiro nome dado a este atributo
    Ideia IdeiaDesign
    Para cada Ideia de domínio I no diagrama
      Se novoNome == I -> obterNome
        Bool existeIdea = Kuaba Facade -> reAceitaIdeia(I,"Operacao")
          Se existeIdea
            IdeiaDesign = KuabaHelper -> obterIdeiaDesign(IdeiaDominio,NomeIdeiaDesign)
          Senão
            IdeiaDesign = KuabaFacade -> criaIdeia()
            KuabaFacade -> criaNovaIdeiaDeDominio(novoNome, ideiadeDesign, IDdaclasse)
            ArgumentController -> ObtemArgumentoParaIdeiaAceita
  Senão
    Para cada Ideia de domínio I faça
      Se I -> Nome == novoNome
        ÉIdeiaDeDominio = true
        ArgumentController -> ObtemArgumentoParaIdeiaAceita
    Se ÉIdeiaDeDominio == false
      Ideia DominioRejeitada = KuabaFacade -> ObtemIdeiaDeDominio(IDAtributo)
      KuabaFacade -> recusaIdeia(DominioRejeitada)
      ArgumentController -> ObtemArgumentoParaIdeiaRecusada
      Ideia IdeiaDominio = KuabaFacade -> criaNovaIdeiaDeDominio(novoNome, IDAtributo)
      ArgumentController -> ObtemArgumentoParaIdeiaAceita
      Ideia classeOwner = DominioRejeitada -> QuestaoSugerida -> ObterOwnership
      DominioAceita -> definirIdeiaQueRespondeQuestao(classeOwner)
      kuabaFacade -> mudaDecisao(IdeiaAceita,classeOwner,true)
      Se houver uma ideia de domínio para nomeAntigo
```

```
    IdeiaRejeitada = KuabaFacade -> obterIdeiaDeDominio(nomeAntigo)
    IdeiaAceita = KuabaFacade -> obterIdeiaDeDominio(novoNome)
    KuabaFacade -> rejeitaOwnershipEAceita(IdeiaRejeitada, IdeiaAceita)
```

Pseudo-código 4.7: Novo método *nameChanged* em *attributeObserver*

```
removedElement(IDElemento)
  eventPump -> removeObserver(IDElemento)
  IdeiaDeDominio = KuabaFacade -> ObtemIdeiaDeDominio(IDElemento)
  configuraDecisaoSubArvore(IdeiaDeDominiorecusada,2)
  ArgumentController -> ObtemArgumentoParaIdeiaRecusada
```

Pseudo-código 4.8: Novo método *removedElement* em *attributeObserver*

4.5 Resultados obtidos

Com a implementação destes algoritmos, a ferramenta passou a exibir um comportamento mais adequado na modificação das decisões tomadas durante o processo de modelagem, sendo possível, por exemplo, recusar ideias de domínios existentes para, logo em seguida, aceita-las novamente.

Outra melhoria perceptível na ferramenta ocorreu na captura das associações. Ao realizar modificações em uma associação como, por exemplo, ao se modificar o nome da associação, agora a KSE realiza a cópia da subárvore de *Design Rationale* da ideia antiga para a nova ideia. Além disso, toda a subárvore da ideia de domínio recusada também tem suas decisões recusadas.

A Figura 4.3 descreve um cenário onde o projetista modela uma classe chamada Editora e outra classe chamada Livro, relacionadas por uma associação no diagrama de classes chamada “Publicado por”. Porém, no processo de modelagem, o projetista decide modificar o nome desta associação para Publica. Ao processar esta modificação, a KSE recusou toda a subárvore de decisões da ideia de domínio “Publicado por” e gerou uma nova ideia de domínio com o novo nome Publica, copiando a subárvore da ideia previamente recusada, porém configurando as decisões como aceitas.

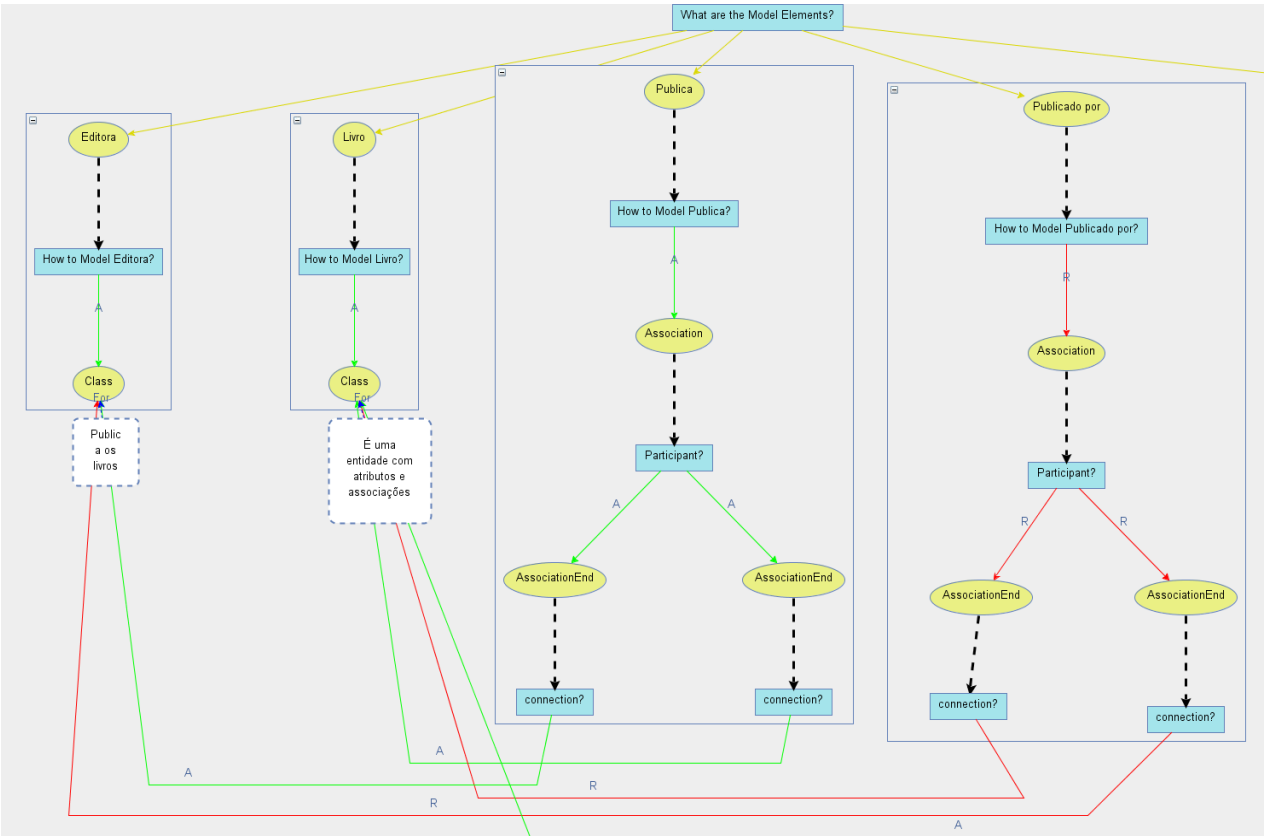

Figura 4.3: A nova ideia de domínio copia a subárvore da ideia de domínio recusada

Capítulo 5

Evolução da KSE: apoio à Captura de Justificativas para Decisões de Design

No capítulo anterior foi discutido o processo de reengenharia de software aplicado na ferramenta KSE, para torná-la mais adequada à ontologia Kuaba e corrigir os problemas encontrados na geração do registro de *Design Rationale*. As mudanças geradas foram necessárias por diversas razões, a maioria delas já explicitadas, com exceção de uma, que será discutida neste capítulo.

Em sua última versão, a ferramenta KSE ainda não possui um recurso que permita a captura das justificativas para as decisões tomadas pelo usuário, conforme previsto na ontologia Kuaba. De acordo com o vocabulário definido nessa ontologia, ilustrado na Figura 2.1, para cada par Questão-Ideia deve ser registrada uma decisão e sua respectiva justificativa. Isto pode ser observado também no exemplo da Figura 2.2, na qual as decisões são ilustradas com rótulos “A” para aceita e “R” para rejeitada. Embora o registro dessas decisões para cada par Questão-Ideia seja necessário, sua representação neste nível de granularidade dificulta a captura de justificativas na ferramenta de design KSE. Para cada decisão registrada deve ser solicitada ao projetista a entrada de uma justificativa. Assim, com o vocabulário atual, o registro de justificativas torna-se uma tarefa extremamente trabalhosa, devido à grande quantidade de decisões que são registradas, podendo interferir negativamente no andamento do trabalho de modelagem do projetista.

5.1 Modificação da Ontologia Kuaba

Para resolver este problema, foi proposta em [Avila & Medeiros, 2014] uma modificação no vocabulário da ontologia Kuaba, que inclui um novo elemento chamado Solução, como ilustra a Figura 5.1. Esse elemento é, essencialmente, um agrupamento de Ideias. Estas Ideias são agrupadas com base nas relações identificadas entre os elementos de raciocínio no grafo de *Design Rationale*. Como alguns desses elementos são gerados a partir do metamodelo da UML, os agrupamentos resultantes seguem as definições dessa linguagem. Por exemplo, o agrupamento de ideias referentes a uma classe em um diagrama, conterá apenas ideias relacionadas a atributos e operações relacionadas a essa classe.

Uma Solução, assim como uma Decisão, pode ser aceita ou recusada. Soluções aceitas são compostas exclusivamente por Ideias aceitas. Do mesmo modo, Soluções recusadas são compostas exclusivamente por Ideias recusadas.

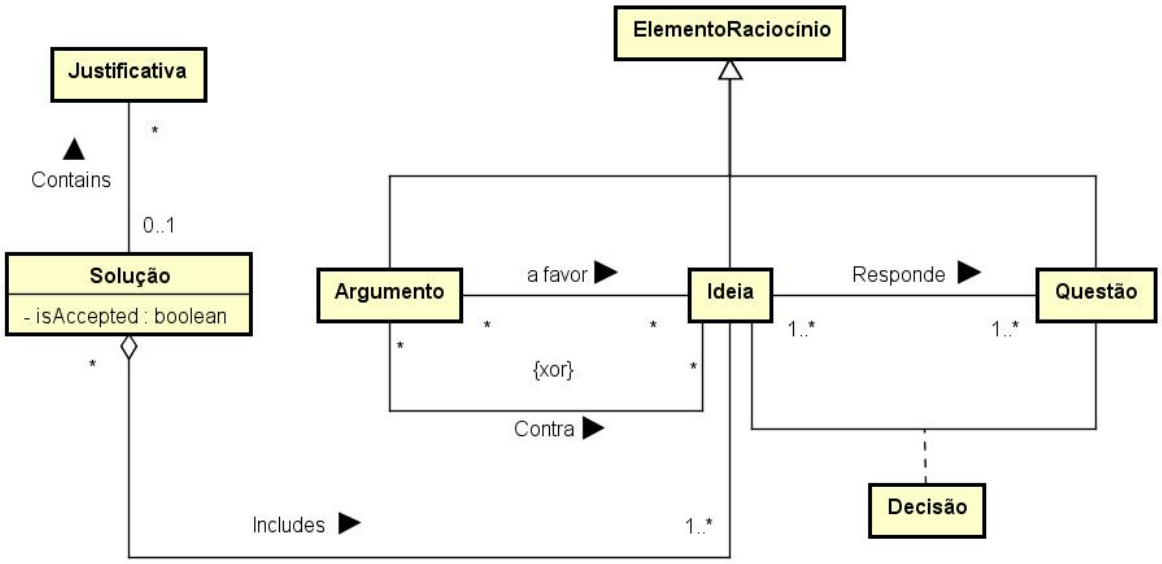

Figura 5.1: Parte do vocabulário da ontologia Kuaba modificado para apoiar o registro de justificativas

Para a KSE, uma mudança no vocabulário da ontologia Kuaba representa uma mudança em um de seus requisitos, que é a instanciação dos elementos dessa ontologia. Mais especificamente neste caso, é necessário que a KSE passe a instanciar o novo elemento da ontologia. Portanto, é necessário realizar uma evolução do software para a adequação a este novo elemento.

A evolução ou manutenção de software é o processo de modificação de um sistema após sua entrega [Sommerville, 2008]. As mudanças podem variar desde simples correções de erros a até mesmo a acomodação de novos requisitos. As mudanças são implementadas nos componentes existentes no sistema e, quando necessário, são adicionados novos componentes. Este tipo de manutenção é chamado de *manutenção evolutiva*, ou simplesmente evolução de software.

5.2 Análise das modificações necessárias

Antes de realizar alterações na ferramenta, é preciso analisa-la para verificar quais

mudanças são necessárias e em quais classes. Para fazer esta análise, pode-se definir uma abordagem *Bottom-up* na estrutura da ferramenta exibida na Figura 3.1.

No nível inferior encontra-se a definição dos elementos da ontologia que são reconhecidos pela KSE. Esta definição está localizada no arquivo de descrição de ontologias no formato OWL, conforme descrito em capítulos anteriores. Assim, é necessário incluir o novo elemento Solução nesse arquivo de definição. De maneira análoga, a representação destes elementos da ontologia no subsistema Kuaba, em forma de classes Java, também deve ser realizada.

Com relação ao processamento do *Design Rationale*, como o elemento Solução é um agrupamento de ideias, suas instancias podem ser geradas automaticamente pela KSE. Assim, é preciso definir algoritmos que sejam capazes de computar a criação/modificação deste novo elemento. Além disso, é necessário definir em qual momento específico do design a KSE deve instanciar o elemento Solução.

Analisando as camadas superiores da arquitetura da KSE, concluiu-se que o elemento Solução deve ser sempre gerado quando já existe um grupo de Ideias que respondem a um problema. Naturalmente, um dos possíveis momentos em que se poderia concluir que o usuário finalizou seu processo de modelagem e gerou as suas soluções, seria quando este decide salvar o diagrama. Neste caso, quando o usuário decide salvar o diagrama, o subsistema Kuaba deve ser notificado e as soluções devem ser geradas.

5.3 Alterações realizadas para criação de Soluções

Nesta subseção serão discutidas as mudanças na estrutura da ontologia Kuaba para incluir este novo elemento. Primeiramente, é necessário declarar o elemento Solução, suas propriedades e relações no arquivo OWL que descreve o vocabulário da ontologia. O Quadro 5.1 mostra esta definição em OWL. O elemento Solução é composto pela relação (*object property*) *contains*, que representa as ideias que formam a solução e pela relação *includes*, que representa a justificativa associada à solução em questão. Além disto, há uma propriedade de dados, chamada *status*, responsável por indicar se esta solução foi aceita ou rejeitada.

```
<Declaration>
    <Class IRI="#Solution"/>
</Declaration>

<Declaration>
    <DataProperty IRI="#accepted"/>
</Declaration>

<Declaration>
    <ObjectProperty IRI="#includes"/>
</Declaration>

<Declaration>
    <ObjectProperty IRI="#contains"/>
</Declaration>

<ObjectPropertyRange>
    <ObjectProperty IRI="#includes"/>
    <Class IRI="#Idea"/>
</ObjectPropertyRange>

<ObjectPropertyRange>
    <ObjectProperty IRI="#contains"/>
    <Class IRI="#Justification"/>
</ObjectPropertyRange>

<ObjectPropertyDomain>
    <ObjectProperty IRI="#includes"/>
    <Class IRI="#Solution"/>
</ObjectPropertyDomain>

<ObjectPropertyDomain>
    <ObjectProperty IRI="#contains"/>
    <Class IRI="#Solution"/>
</ObjectPropertyDomain>

<DataPropertyRange>
    <DataProperty IRI="#accepted"/>
    <Datatype abbreviatedIRI="xsd:boolean"/>
</DataPropertyRange>
```

Código 5.1: Elementos e relações incluídos na ontologia Kuaba em OWL

Na sequência, é necessário que a KSE possa reconhecer e processar este elemento. Para fazê-lo, é necessário implementar classes e interfaces Java correspondentes a este elemento. As classes implementam as responsabilidades que são descritas na interface que representa este elemento.

```
public interface Solution extends KuabaElement{
    void addContains(Justification newJustification);
    void addIncludes(Idea newIdea);
    boolean hasContains();
    boolean hasIncludes();
    void removeIncludes(Idea oldIdea);
    Collection<Justification> getContains();
    Collection<Idea> getIncludes();
        boolean hasAccepted();
        boolean getAccepted();
        void setAccepted(boolean accepted);
        }
```

Código 5.2: Descrição da interface do elemento Solução

O processamento deste elemento é dependente da definição do seu modo de utilização. Ou seja, baseando-se no conceito do elemento Solução, é necessário verificar quais classes são afetadas para que se realize a sua instanciação e manipulação no *Design Rationale*. Como dito anteriormente, o momento sugerido para realizar este tratamento seria durante a etapa de registro do *Design Rationale*.

Dentro da arquitetura da KSE, ao enviar um comando para armazenar o diagrama UML, o subsistema recebe uma notificação para também gravar o *Design Rationale*. Para gravar o registro de *Design Rationale*, o subsistema Kuaba delega à classe *kuabaFacade* a responsabilidade de processar o grafo existente e, a partir das descrições existentes no arquivo OWL, representado pelas interfaces e classes, enviar um comando para uma classe no pacote *Repository* para gerar o arquivo OWL com a descrição do grafo de *Design Rationale*.

Devido à estrutura descrita, um momento adequado para realizar o processamento do elemento *Solução* seria logo antes da geração do arquivo OWL e após a notificação do subsistema Kuaba da tarefa de gravar o modelo, portanto na classe *kuabaFacade*, mais precisamente no método *saveSession*, ilustrado no Quadro 5.3. Este método cria a decisão para as ideias de domínio e logo em seguida delega à classe *RepositoryGateway* a responsabilidade de gerar os arquivos OWL.

```
public boolean saveSession(KuabaRepository kr, File destination){
   Question whatElements = modelRepository().getQuestion(Question.ROOT_QUESTION_ID);
   List<Idea> consideredIdeas = this.session.unitOfWork().getConsideredIdeas();
   for (Idea idea : consideredIdeas)
      this.makeDecision(whatElements, idea, true);
   List<Idea> rejectedIdeas = this.session.unitOfWork().getRejectedIdeas();
   for (Idea idea2 : rejectedIdeas)
      this.makeDecision(whatElements, idea2, false);
   return KuabaSubsystem.gateway.save(kr, destination);
```

Código. 5.3: Método *saveSession*

Com estas constatações, decidiu-se por adicionar o processamento das soluções no método *saveSession*, para que, ao salvar o arquivo OWL, as soluções estejam processadas, tanto para os agrupamentos que formam soluções aceitas quando agrupamentos que formam soluções rejeitadas.

Em [Avila & Medeiros 2014] é definido um algoritmo, independente de metamodelo, que realiza o agrupamento de ideias em soluções, sejam estas aceitas ou recusadas. O algoritmo se baseia em ideias de design centrais para gerar as soluções. Ou seja, para cada ideia de design que responde uma questão levantada por uma ideia de domínio, verifica-se quais outras questões são endereçadas por esta ideia de design e se esta ideia central também sugere alguma questão, que pode ser endereçada por outras ideias de design. A partir destas relações, realiza-se o agrupamento das ideias, gerando as soluções. O pseudo-código 5.1 contém abaixo contém a definição do algoritmo utilizado para realizar este agrupamento.

varrerSubArvoreAceita(Ideia X, Solucao A)

Marcar X como elemento do Solução A

Se X sugere uma questão Q (X suggests Q) e Q tem uma decisão D (Q hasDecision D), onde D conclui X (X isConcludedBy D) e D é aceita (D isAccepted => true), **então**

Marcar questão Q como elemento da solução A;

Enquanto existir uma ideia de design i que responde a questão Q (i adress => Q) e Q tem uma decisão D (Q hasDecision D), onde D conclui X (X isConcludedBy D) e Dé aceita (D isAccepted => true)

FuncaoVarrerSubArvoreAceita(i, A);

FuncaoVarrerSubArvoreRejeitada(Ideia de Design X, Solução A)

Marcar X como elemento da solução A;

Se X sugere uma questão Q (X suggests Q) e Q tem uma decisão D (Q hasDecision D), onde D conclui X (X isConcludedBy D) e D é rejeitada (D isAccepted => false), **então**

Marcar questão Q como elemento do solução A;

Enquanto existir uma ideia de design i que responde a questão Q (i adress => Q) e Q tem uma decisão D (Q hasDecision D), onde D conclui X (X isConcludedBy D) e Dé rejeitada (D isAccepted => false)

FuncaoVarrerSubArvoreRejeitada(i, A);

criarSolucoes()

Para cada ideia de design X que responda a uma questão Q e Q seja sugerida por uma ideia de domínio Z

Adicione X à lista L

Para cada ideia de design X na lista L

Para cada ideia de design i que responde uma questão Q que seja sugerida por X e Q tem uma Decisão D, onde D conclui i e D é aceita

cont ++;

Se cont >=2

Criar solução A

Marcar X como Ideia de Design Central da solução A

FuncaoVarrerSubArvoreAceita(X, A)

Para cada Ideia de design i marcada como pertencente à solução A

Se i responde uma questão Q e Q que é sugerida por uma Ideia de domínio Z

Marque Q como pertencente à solução A

Marque Z como pertencente à solução A

Se i pertence à lista L e para toda questão Q a qual i responde Q é marcada como pertencente à solução A

Tirar i da lista L

Para cada ideia de design i que responde uma questão Q que seja sugerida por X e Q tem uma decisão Di, onde Di conclui i e Di é rejeitada

Criar solução A

Marque X como Ideia de Design Central da solução A

Marque Q como pertencente à solução A

FuncaoVarrerSubArvoreRejeitada(i, A)

Para cada Ideia de design i marcada como pertencente à solução A

Se i responde uma questão Q que é sugerida por uma Ideia de domínio Z

Marque Q como pertencente à solução A

Marque Z como pertencente à solução A

Se i pertence à lista L e para toda questão Q a qual i responde Q é marcada como pertencente à solução A

Tirar i da lista L

Se X responde a uma questão Q1 e Q1 é sugerida por uma ideia de domínio Z, onde Z responde a uma questão Q2 e Q2 tem uma decisão D onde D conclui Z e D é rejeitada

Criar solução A

Marcar X como Ideia de Design Central do agrupamento A

FuncaoVarrerSubArvoreRejeitada(X, A)

Para cada Ideia de design i marcada como pertencente à solução A

Se i responde uma questão Q e Q é sugerida por uma Ideia de domínio Z

Marque Q como pertencente à solução A

Marque Z como pertencente à solução A

Se i pertence à lista L e para toda questão Q a qual i responde Q é marcada como pertencente à solução A

Tirar i da lista L

Se X responde a duas ou mais questões **Qi**, onde cada questão **Qi** tem uma decisão **Di** e **Di** conclui X, **Di** é aceita e as questões **Qi** são iguais

Criar solução A

Marque X como Ideia de Design Central da solução A

Para cada questão **Qi** igual

Marque **Qi** como pertencente ao agrupamento A

Marque a ideia de design Y que sugere **Qi** como pertencente à solução A

Para cada ideia de design i pertencente à solução A

FuncaoVarrerSubArvoreAceita(i, A)

Para cada Ideia de design i marcada como pertencente à solução A

Se i responde uma questão Q e Q que é sugerida por uma Ideia de domínio Z

Marque Q como pertencente à solução A

Marque Z como pertencente à solução A

Se i pertence à lista L e para toda questão Q a qual i responde, Q é marcada como pertencente à solução A

Tirar i da lista L

Para cada ideia de design X da lista L, faça

Se X responde a uma única questão Q onde Q é sugerida por uma ideia de design Y, onde Q tem uma decisão D e D conclui X, D é aceita

Criar solução A

Marque X como Ideia de Design Central do agrupamento A

Marque Q como pertencente à solução A

Marque Y como pertencente à solução A

Para cada Ideia de design i marcada como pertencente à solução A

FuncaoVarrerSubArvoreAceita (i, A)

Para cada Ideia de design i marcada como pertencente à solução A

```
                    Se i responde uma questão Q e Q que é sugerida por uma Ideia de domínio Z
                        Marque Q como pertencente à solução A
                        Marque Z como pertencente à solução A
                    Se i pertence à lista L e para toda questão Q a qual i responde, Q é marcada como pertencente à solução A
                        Tirar i da lista L
                Enquanto houver uma ideia de design X na lista L
                    Criar solução A
                    Marque X como Ideia de Design Central da solução A
                    FuncaoVarrerSubArvoreAceita(X, A)
                Para cada Ideia de design i marcada como pertencente à solução A
                Se i responde uma questão Q e Q que é sugerida por uma Ideia de domínio Z
                    Marque Q como pertencente à solução A
                    Marque Z como pertencente à solução A
                Se i pertence à lista L e para toda questão Q a qual i responde, Q é marcada como pertencente à solução A
                    Tirar i da lista L
```

Pseudo-Código 5.1: Funções que realizam os agrupamentos de ideias em soluções

5.4 Alterações realizadas para apoiar a captura de justificativas

Após o processamento das soluções, é necessário realizar a captura das suas respectivas justificativas. Para realizar esta captura, decidiu-se por exibir uma tela ao usuário requisitando esta informação. Esta tela exibe ao projetista uma pergunta contendo as ideias de domínio agrupadas e um espaço para inserir a justificativa. Optou-se por elaborar as perguntas em texto corrido, para serem mais amigáveis ao usuário.

A Figura 5.2 reproduz a tela para inserção da justificativa para uma solução que contém as ideias de domínio Pessoa, modelada como uma classe, Nome, modelada como atributo, e getNome, modelada como operação.

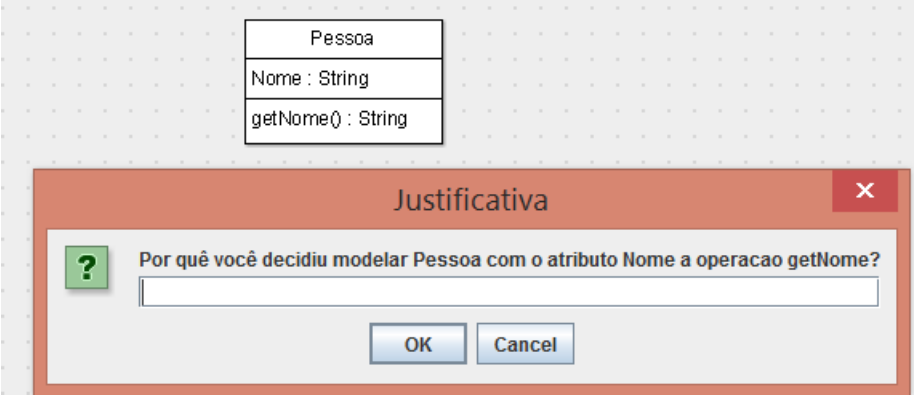

Figura 5.2: Tela para inserção da justificativa para uma solução aceita entre classe, atributo e operação.

A Figura 5.3 exibe a tela para captura de uma solução rejeitada. Neste caso a pergunta endereçada ao projetista é modificada. Pergunta-se o porquê de o projetista ter desistido de realizar a modelagem original.

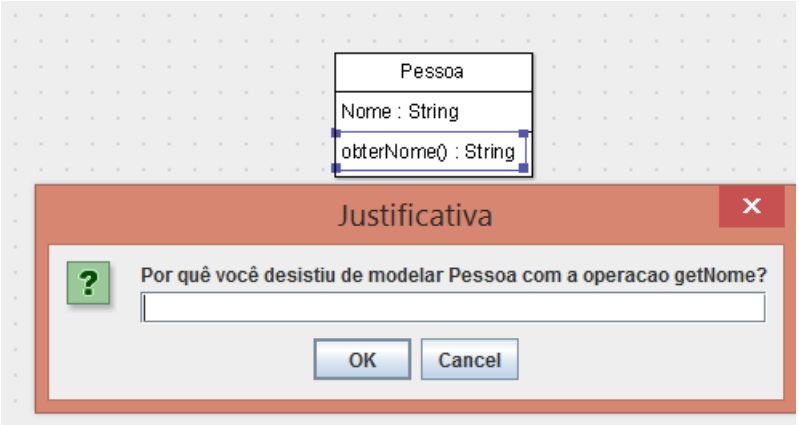

Figura 5.3: Tela para inserção da justificativa para uma solução rejeitada entre classe, atributo e operação.

A Figura 5.4 contém a tela de justificativa para uma solução que contenha um agrupamento de ideias relacionadas a uma associação e que foi aceita.

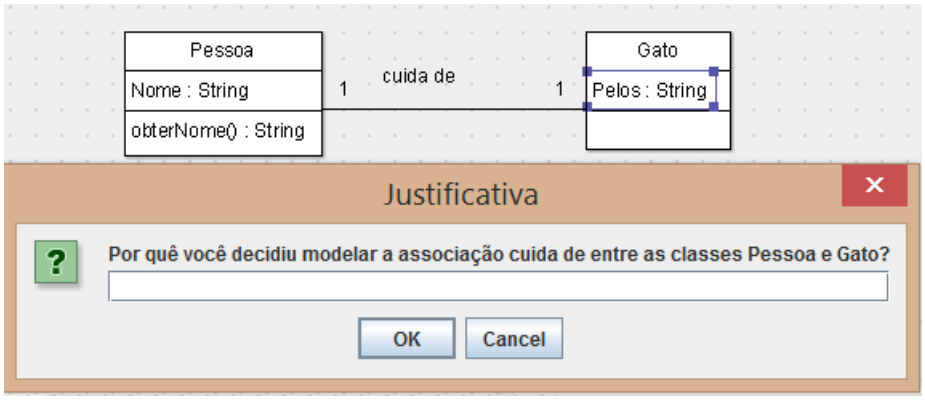

Figura 5.4: Tela para inserção da justificativa para uma solução aceita entre os elementos de uma associação.

A Figura 5.5 contém a tela de justificativa para uma solução que contenha um agrupamento de ideias relacionadas a uma associação e que foi rejeitada.

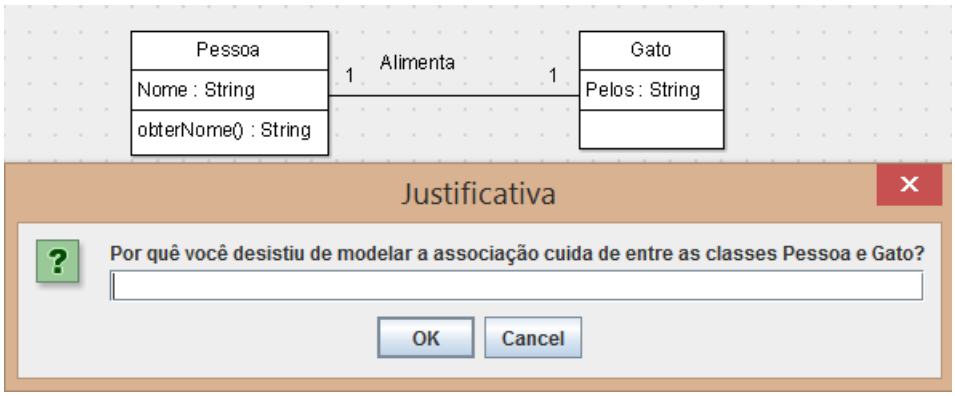

Figura 5.5: Tela para inserção da justificativa para uma solução rejeitada entre os elementos de uma associação.

Capítulo 6

Verificação dos resultados

Para verificar os resultados obtidos através das alterações aplicadas, foi realizada uma simulação de uso da ferramenta KSE. Esta simulação consiste na produção de um diagrama de classes para modelar o domínio de uma livraria. Neste domínio, os livros são publicados por editoras específicas e possuem autores. Os livros podem ser publicados por apenas uma editora, mas as editoras podem publicar diversos livros. Os autores podem ter seus livros publicados por diversas editoras também.

Primeiramente, o projetista decide modelar as propriedades da classe Livro, incluindo os atributos Nome e Autor, porém após análise, decide que o campo Autor não deve ser um atributo, removendo este elemento do diagrama. Esta solução gera o seguinte grafo de DR:

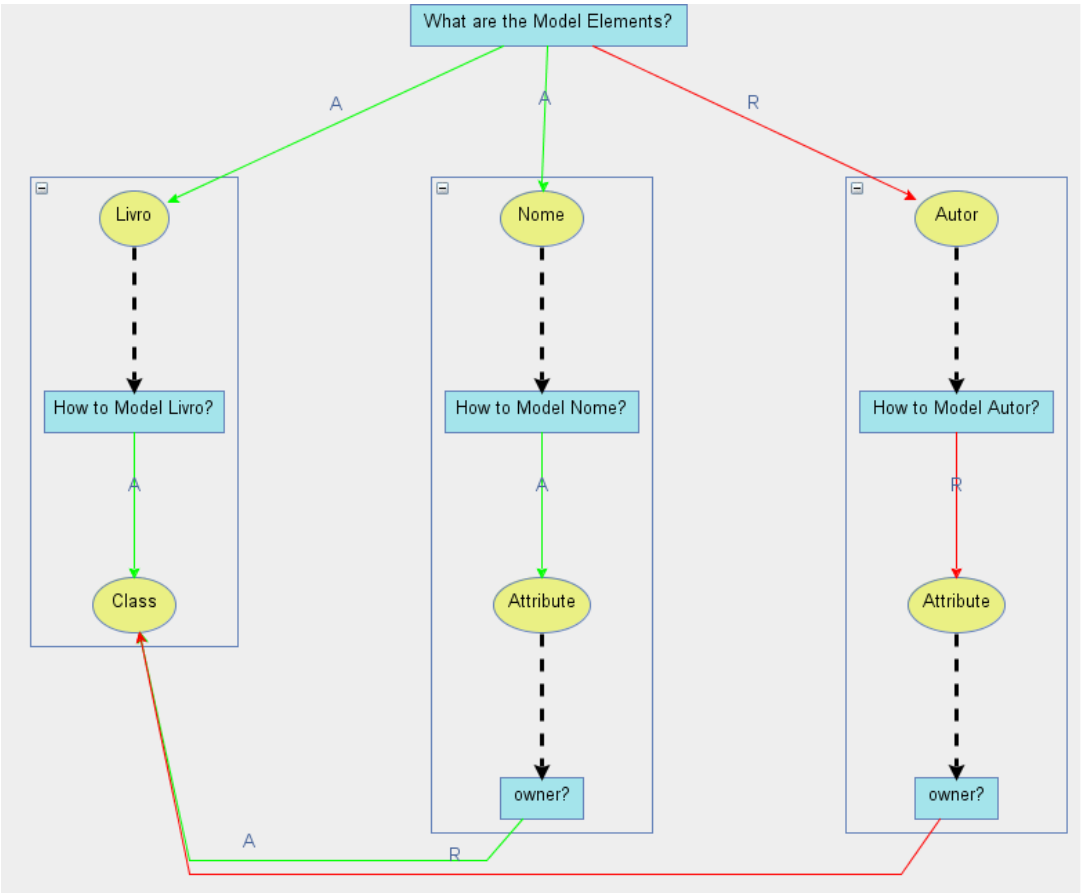

Figura 6.1: Grafo de *Design Rationale* gerado ao modelar a classe Livro.

Após verificar o resultado de sua modelagem, o projetista decide salvar o diagrama. A KSE executa o algoritmo para agrupamento de ideias em soluções e requisita uma justificativa ao projetista sobre porque Livro e Nome formam uma solução aceita.

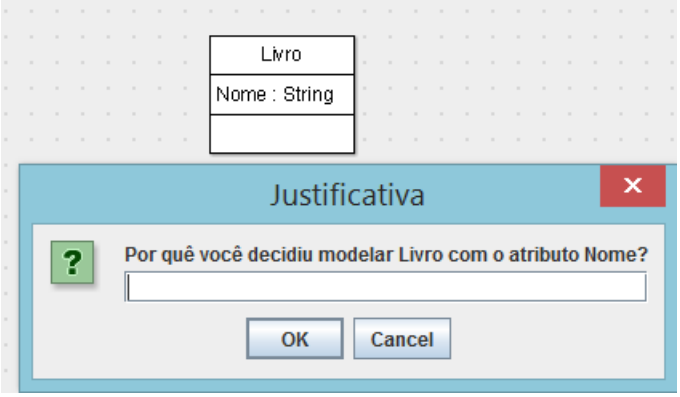

Figura 6.2: KSE requisita justificativa para uma solução aceita

Na sequência, a KSE requisita novamente uma justificativa, porém para a solução que foi rejeitada pelo projetista, formada pela classe Livro e o atributo Autor, que foi removido do diagrama.

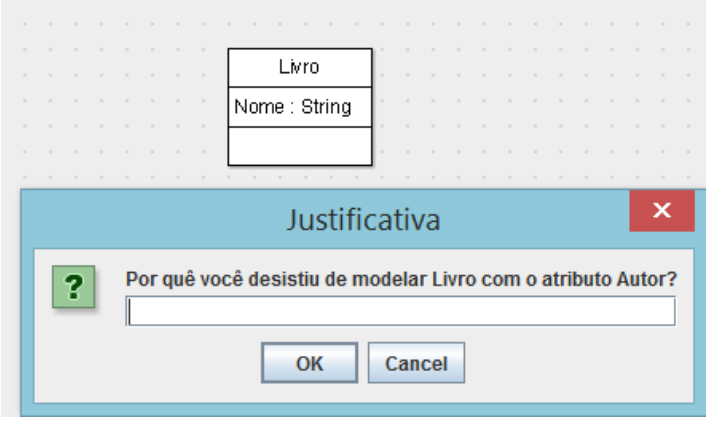

Figura 6.3: KSE requisita justificativa para solução rejeitada.

Na próxima sequência de telas, o projetista decide recomeçar a modelagem partindo de associações entre elementos do domínio, que são Livro, Autor e Editora. A Figura 6.4 ilustra uma possível modelagem para este cenário.

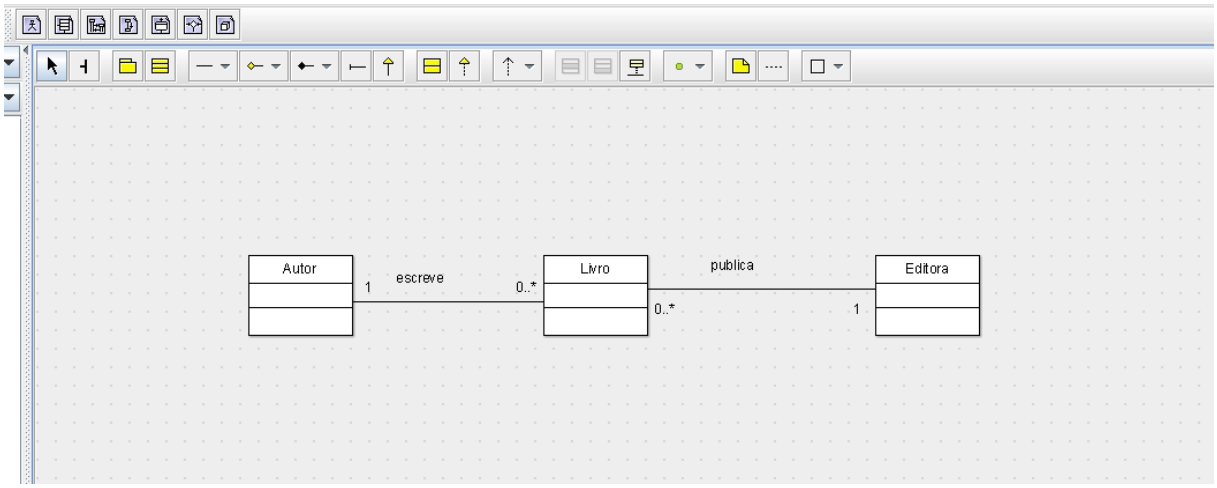

Figura 6.4: Exemplo de diagrama de classes para o domínio de uma livraria

Nesse exemplo, o projetista cria duas associações relacionando os conceitos do domínio, “Autor escreve Livro” e “Editora publica Livro”. A Figura 6.5 exibe o grafo de *Design Rationale* gerado pela KSE para a modelagem realizada.

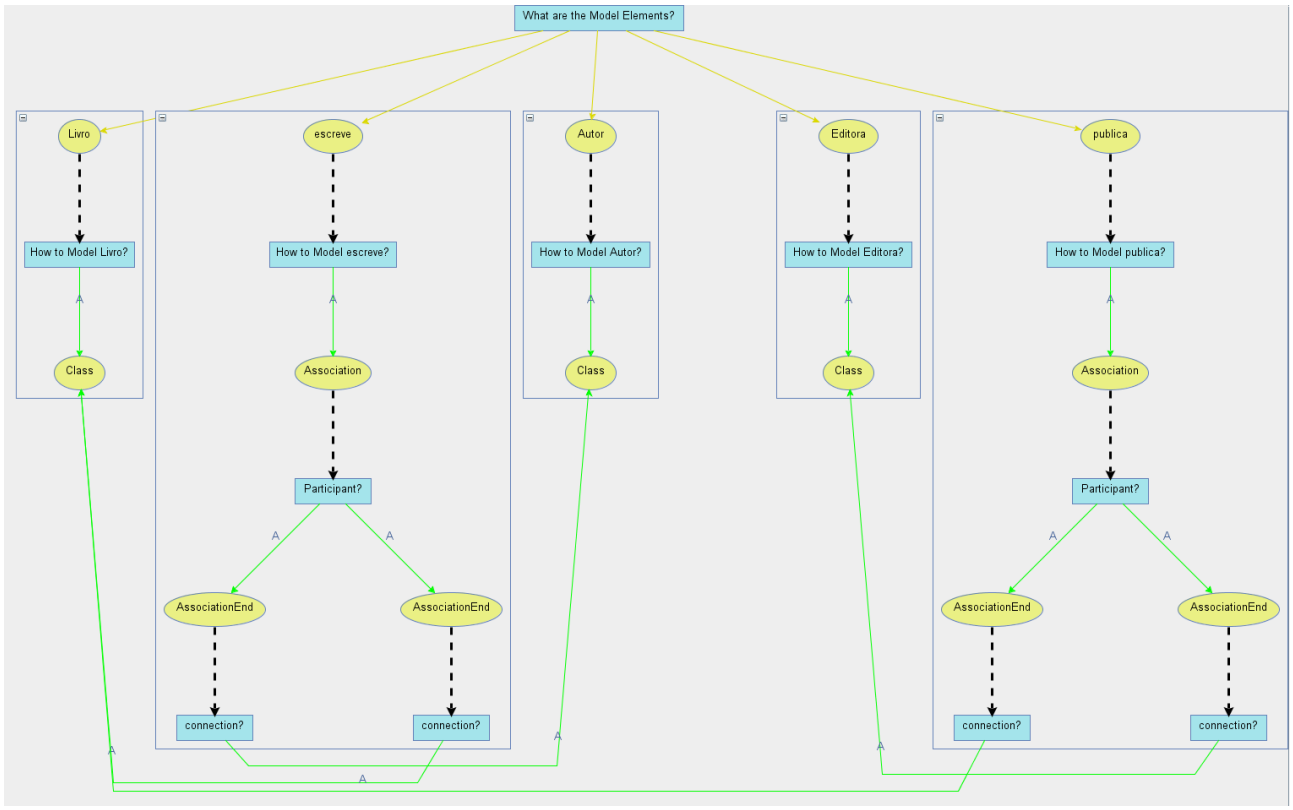

Figura 6.5: Grafo de *Design Rationale* gerado pela KSE para o diagrama de classes da livraria

Neste ponto, a KSE gerou todas as questões, ideias e decisões, similar à implementação original, corretamente. Na próxima etapa, o projetista desiste de modelar Autor como uma classe, então remove a classe correspondente do diagrama.

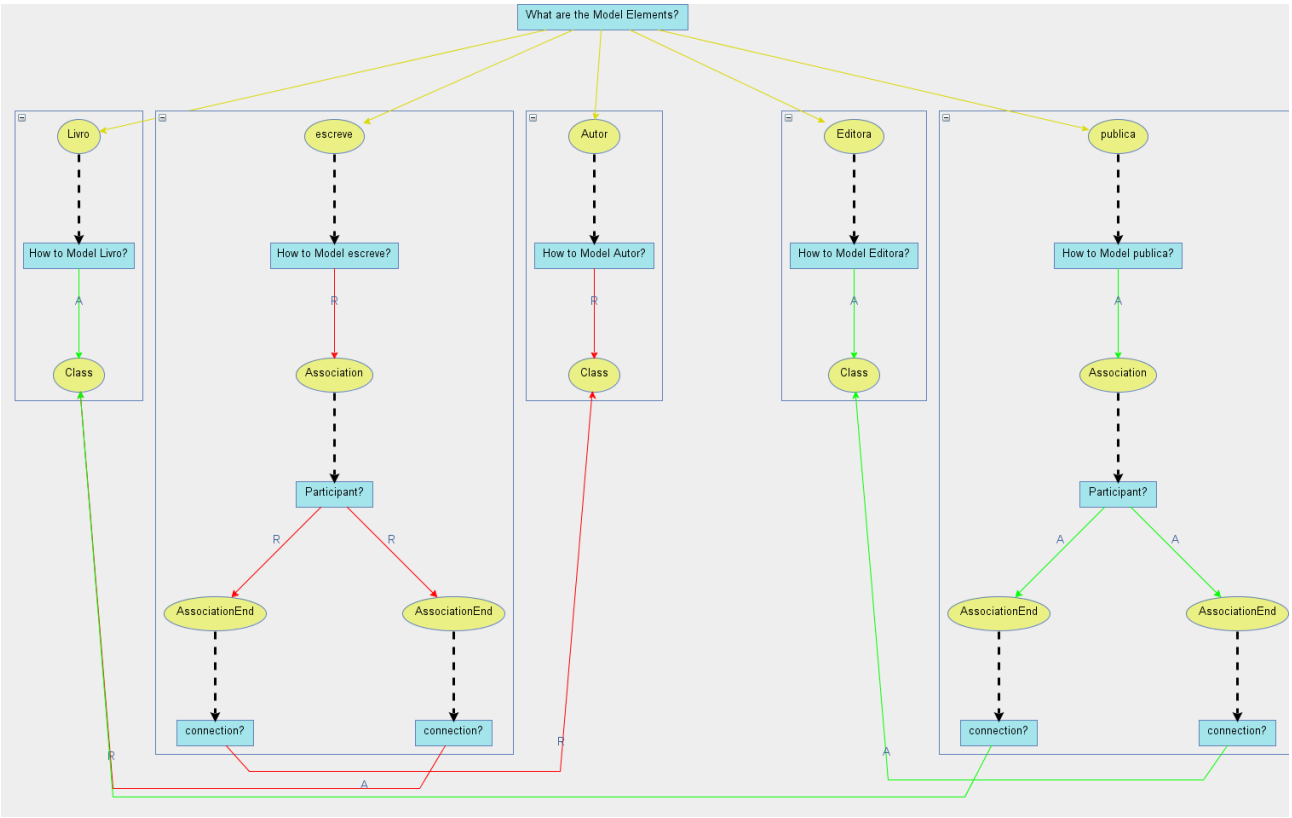

Figura 6.6: As ideias de domínio Autor e Escreve, e suas respectivas ideias de design são marcadas como rejeitadas no grafo de *Design Rationale*

Como resultado dessa alteração, a ideia de domínio Autor tem sua sub-árvore rejeitada, ou seja, as decisões para as ideias de design relacionadas a essa ideia de domínio são gravadas como rejeitadas, como ilustra a Figura 6.6. De maneira análoga, a ideia de domínio para a associação “Escreve” também tem sua sub-árvore rejeitada, pois a própria associação também não existe mais. A próxima etapa simulada foi a criação de Autor como um atributo de Livro, como ilustra a Figura 6.7:

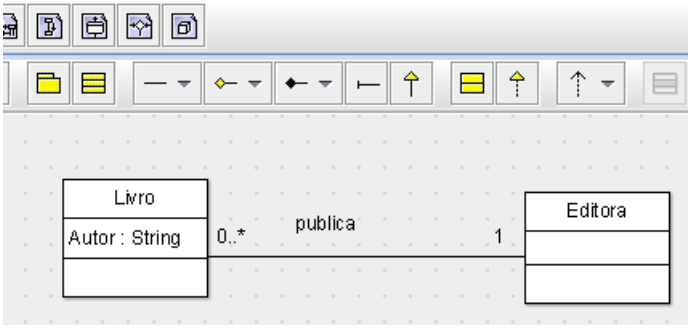

Figura 6.7: Mudança da ideia de domínio Autor de Classe para Atributo

A Figura 6.8 mostra o registro de uma nova ideia de design *Attribute* que responde à questão sugerida pela ideia de domínio Autor. Anteriormente, a KSE criaria uma nova ideia de domínio, com o mesmo nome de uma ideia já existente, para associar a ideia de design, porém após o processo de Reengenharia a ferramenta KSE passou a gerar corretamente o grafo de *Design Rationale* de acordo com a ontologia Kuaba.

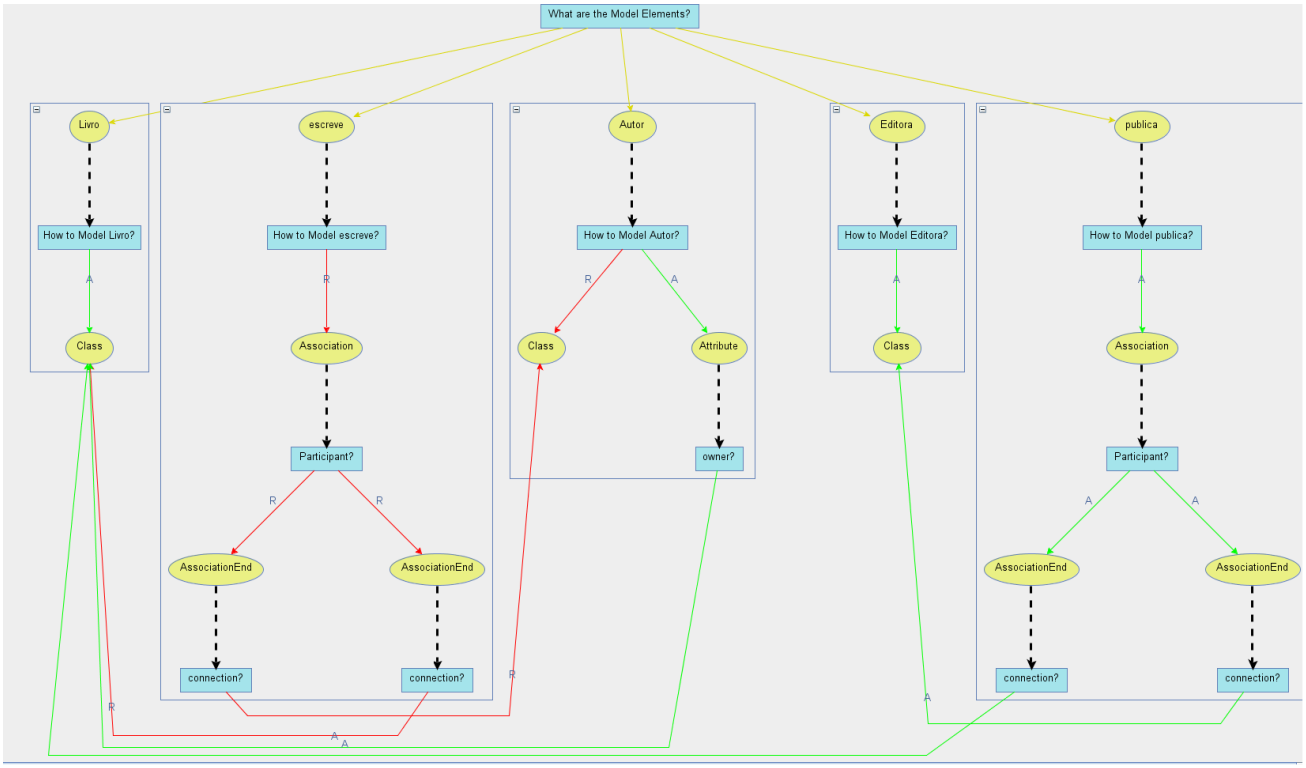

Figura 6.8: Grafo de *Design Rationale* modificado com a ideia de design *Attribute* na subárvore da ideia Autor

No próximo passo da verificação, foi modificado o nome da associação entre as classes Livro e Editora de “publica” para “Distribui”, como ilustra a Figura 6.9:

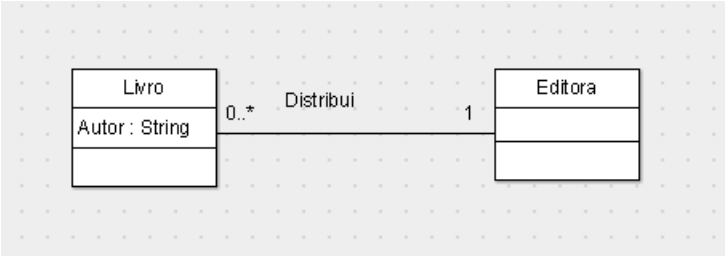

Figura 6.9: Nome da associação entre “Livro” e “Editora” modificado para “distribui”

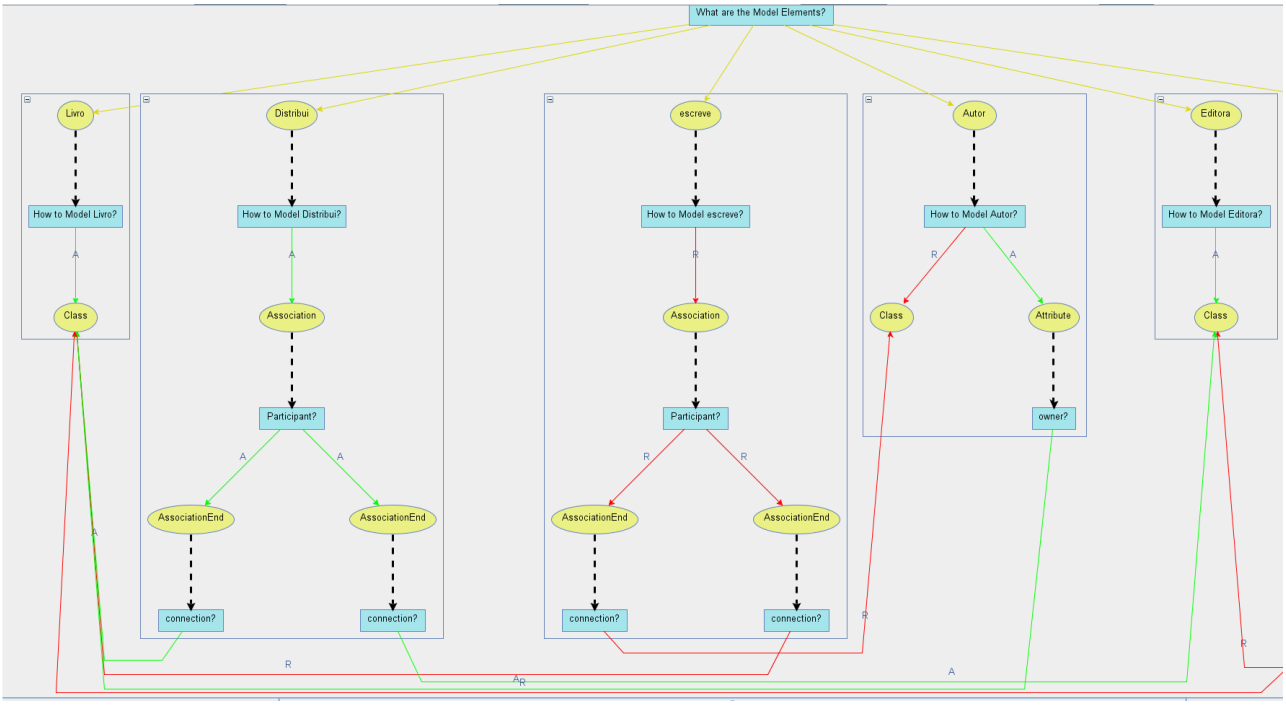

Figura 6.10: Grafo de *Design Rationale* modificado após a alteração do nome da associação.

Ao modificar o nome da associação, a KSE criou uma nova ideia de domínio, com o novo nome da associação, “distribui”, e atribuiu uma sub-árvore de *Design Rationale* similar à ideia de domínio rejeitada, com as decisões consideradas aceitas. Já a ideia de domínio original “publica” teve sua sub-árvore completamente rejeitada, conforme visto na Figura 6.10. Na próxima etapa, considera-se que o projetista deseja concluir sua modelagem e já

obteve um modelo que atende às suas necessidades. Para concluir, então, ele grava o diagrama.

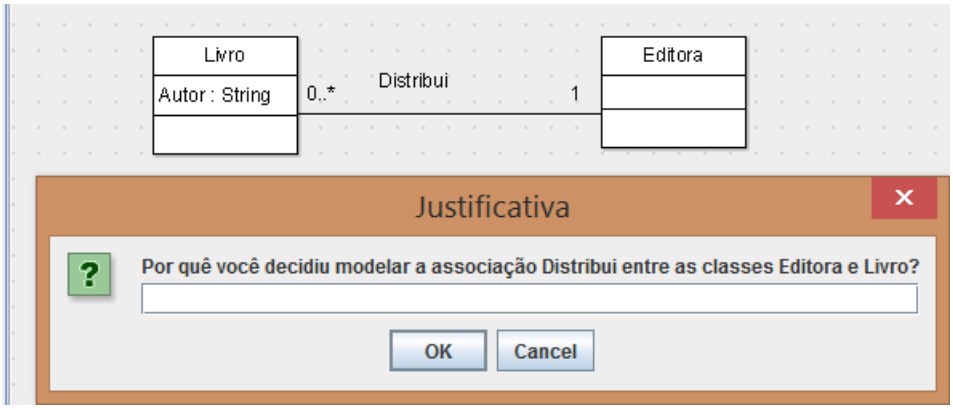

Figura 6.11: KSE requisita a justificativa para a solução com “Distribui, “Editora” e “Livro”

A Figura 6.11, mostra que, ao salvar o diagrama, a KSE requisitou justificativa para uma solução aceita que foi gerada utilizando o algoritmo descrito em 5.1. A justificativa foi requisitada em uma tela que contém uma pergunta que exibe as ideias que formam o agrupamento em questão, Livro, Editora e Distribui.

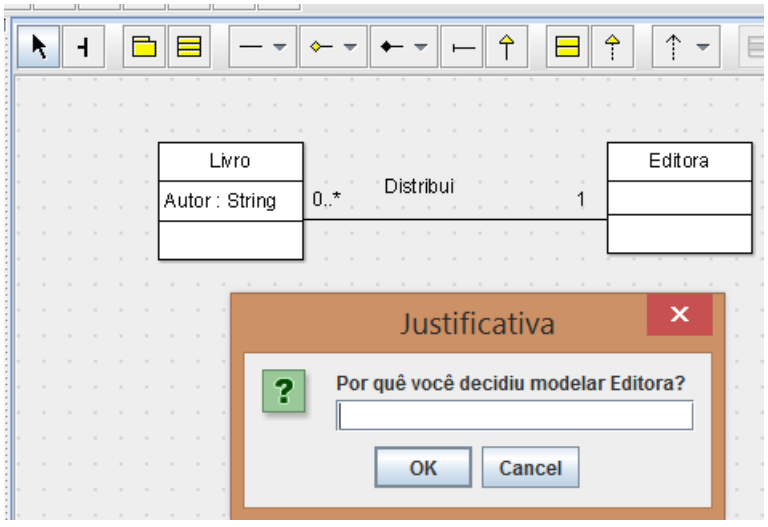

Figura 6.12: KSE requisita justificativa para a solução com a ideia Editora

Em seguida a KSE exibiu uma nova janela, requisitando outra justificativa, agora para outro agrupamento de ideias, com Livro e Autor.

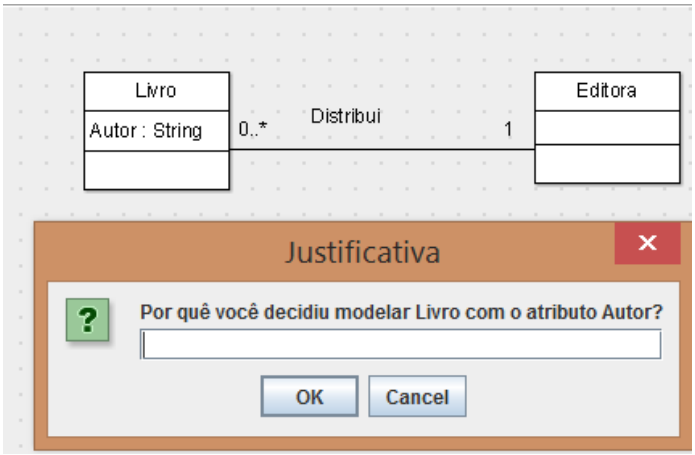

Figura 6.13: KSE requisita justificativa para solução com as ideias “Livro” e “Autor”.

A próxima janela exibida pela KSE requisita ao usuário a inserção de uma justificativa para uma solução que foi rejeitada, contendo as ideias que representam uma associação entre Livro e Autor.

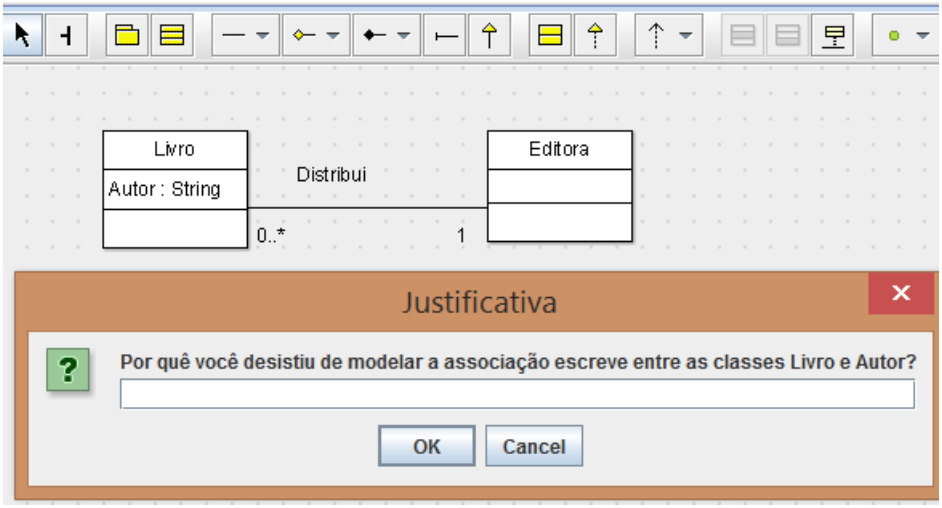

Figura 6.14: KSE requisita justificativa para uma solução rejeitadaPor fim, a Figura 6.15 exibe a última tela para inserção de justificativa para uma associação rejeitada, contendo as ideias Editora, Livro e publica.

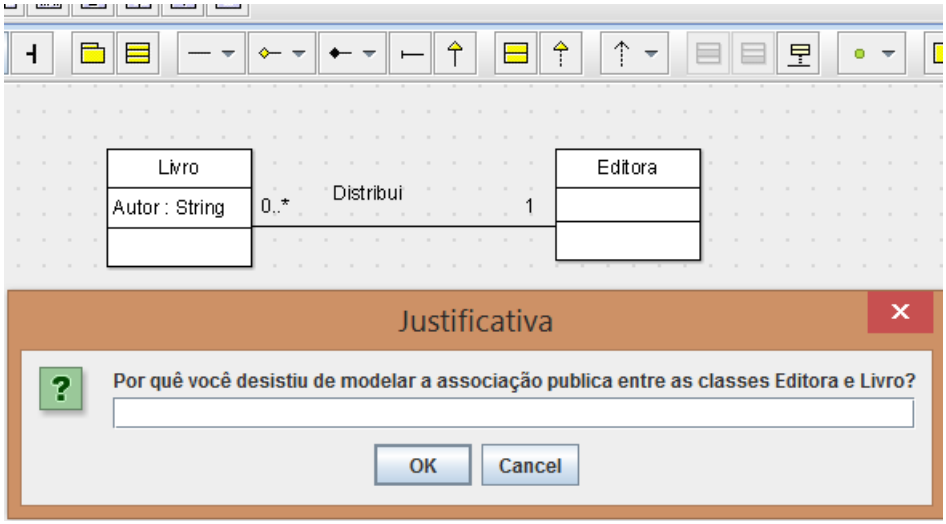

Figura 6.15: KSE requisita justificativa para solução rejeitada contendo as ideias publica, Editora e Livro

A Figura 6.16 contém o arquivo OWL gerado após a gravação do diagrama de classes. Verifica-se que as soluções foram preenchidas com ideias e justificativas conforme esperado e os status das soluções foram configurados corretamente. No campo *accepted*, soluções rejeitadas foram configuradas como *false* e soluções aceitas foram configuradas como *true*.

```
<owl:NamedIndividual rdf:about="owlapi:ontology1204664747741664#_210208">
    <rdf:type rdf:resource="http://www.progen.kilu.de/DesignRationale/KuabaOntology.owl#Solution"/>
    <KuabaOntology:accepted rdf:datatype="http://www.w3.org/2001/XMLSchema#string">true</KuabaOntology:accepted>
    <KuabaOntology:includes rdf:resource="owlapi:ontology1204664747741664#93094ad7-4964-4f66-94ce-5551bee99f01"/>
    <KuabaOntology:includes rdf:resource="owlapi:ontology1204664747741664#_000000000000101B_210194"/>
    <KuabaOntology:includes rdf:resource="owlapi:ontology1204664747741664#_000000000000101C_210196"/>
    <KuabaOntology:contains rdf:resource="owlapi:ontology1204664747741664#_210211"/>
    <KuabaOntology:includes rdf:resource="owlapi:ontology1204664747741664#dc5d0077-2117-465c-baac-812d56306024"/>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="owlapi:ontology1204664747741664#_210211">
    <rdf:type rdf:resource="http://www.progen.kilu.de/DesignRationale/KuabaOntology.owl#Justification"/>
    <KuabaOntology:hasText rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        Livro e Nome formam uma entidade do domínio Livraria
    </KuabaOntology:hasText>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="owlapi:ontology1204664747741664#_210207">
    <rdf:type rdf:resource="http://www.progen.kilu.de/DesignRationale/KuabaOntology.owl#Solution"/>
    <KuabaOntology:accepted rdf:datatype="http://www.w3.org/2001/XMLSchema#string">false</KuabaOntology:accepted>
    <KuabaOntology:includes rdf:resource="owlapi:ontology1204664747741664#93094ad7-4964-4f66-94ce-5551bee99f01"/>
    <KuabaOntology:includes rdf:resource="owlapi:ontology1204664747741664#_000000000000101B_210194"/>
    <KuabaOntology:includes rdf:resource="owlapi:ontology1204664747741664#_000000000000101F_210200"/>
    <KuabaOntology:contains rdf:resource="owlapi:ontology1204664747741664#_210210"/>
    <KuabaOntology:includes rdf:resource="owlapi:ontology1204664747741664#a2a0cd47-3102-4ed7-a87c-0abe4539da0b"/>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="owlapi:ontology1204664747741664#_210210">
    <rdf:type rdf:resource="http://www.progen.kilu.de/DesignRationale/KuabaOntology.owl#Justification"/>
    <KuabaOntology:hasText rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        O Autor deve ser uma entidade do domínio
    </KuabaOntology:hasText>
</owl:NamedIndividual>
```

Figura 6.16: Arquivo OWL com a descrição do *Design Rationale*

Capítulo 7

Conclusão e Trabalhos Futuros

Neste trabalho foram apresentadas melhorias desenvolvidas na ferramenta KSE. Esta ferramenta CASE realiza a captura de *Design Rationale* tendo como base a abordagem Kuaba. As melhorias aplicadas na KSE foram: a utilização de técnicas de reengenharia de software para remover o acoplamento durante a captura de *Design Rationale*, e corrigir o registro das decisões. Por fim, foi realizada uma evolução na ferramenta para incorporar o agrupamento de Ideias a partir de um novo elemento da ontologia, chamado Solução.

Nos testes realizados, as modificações realizadas se mostraram satisfatórias. A ferramenta apresentou melhorias no tratamento das decisões geradas para as ideias que são criadas e modificadas no diagrama, representando, em tempo real, os valores esperados. Isso pode ser atribuído às técnicas de reengenharia aplicadas, que puderam indicar as falhas presentes na ferramenta e os novos algoritmos implementados para corrigi-las. Os padrões de reengenharia de software utilizados se mostraram bem sucedidos na descrição e obtenção da estrutura lógica de organização da ferramenta.

Com relação à evolução da KSE, a implementação realizada permitiu à ferramenta realizar a captura de justificativas, um elemento da ontologia Kuaba que não estava sendo tratado. Além disso, ao agrupar as ideias em soluções, a ferramenta diminui a quantidade de solicitações que faz ao projetista para requisitar as justificativas, tornando-a menos intrusiva. É preciso verificar, porém, se a geração das soluções somente no momento de salvar o diagrama é adequada ou se haveria outro momento mais adequado para processar estes agrupamentos.

Entre os problemas identificados durante a realização deste trabalho, encontram-se:

- Os eventos disparados pela ferramenta ArgoUML durante a modelagem do diagrama impediram a captura de *Design Rationale* para outros elementos do diagrama de classes UML, como por exemplo: a relação de Hierarquia entre classes, ou a implementação de interfaces.

- A dependência da ferramenta KSE aos eventos disparados pela ArgoUML, que gera uma dependência do subsistema Kuaba à ferramenta de modelagem.

- A representação gráfica do par questão-ideia, para ideias de design que respondem à uma mesma questão nem sempre é realizada. É preciso reavaliar o modo de representação destes elementos na ferramenta.

- Falta de documentação detalhada com relação à estrutura do código e suas funções.

Com base nos problemas identificados e na experiência com a ferramenta, recomenda-se, como trabalho futuro, um aperfeiçoamento do desacoplamento do subsistema Kuaba à ferramenta ArgoUML, ou seja, melhorar a implementação do padrão de projeto *Adapter*, na integração entre a ferramenta de modelagem e o subsistema, conforme previsto na arquitetura original do sistema.

Atualmente a captura de argumentos na KSE é realizada sempre que um elemento do tipo argumento é gerado. Recomenda-se como trabalho futuro definir uma nova metodologia para se capturar estes elementos, baseando-se na captura das justificativas para as soluções, descritas neste trabalho.

Também como trabalho futuro, recomenda-se ajustes na representação gráfica do *Design Rationale* na ferramenta, assim como uma melhor documentação, para facilitar seu processo de manutenção e evolução.

Referências Bibliográficas

1. ArgoUML User Manual (2008). http://argouml-stats.tigris.org/documentation/manual-0.26/. Acessado em 10/07/2016.

2. Avila, G. Q. & Medeiros A. P. (2014). *Extensão do Modelo de Representação de Design Rationale Kuaba para permitir a Captura de Justificativas para Decisões de Design*. Relatório de Projeto de Pesquisa. Instituto de Ciência e Tecnologia. Universidade Federal Fluminense.

3. Diniz, B. (2013). Extensão da ferramenta KSE para apoiar reúso de design de software utilizando rationale. Monografia de Conclusão de Curso de Ciência da Computação. Instituto de Ciência e Tecnologia. Universidade Federal Fluminense. 45 Pág.

4. Demeyer, S. & Ducasse, S. & Nierstraz, O. (2008). *Object-Oriented Reengineering Patterns*. Square Bracket Associates.

5. Gamma, E., Helm, R., Jonhson, R., & Vlissides, J. (1995). *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

6. Guarino, N., Oberle, D., Steffen, S. (2009). *Handbook of ontologies*. 2º Edition, Springer.

7. KUNZ, W.; RITTEL, H. W. J. Issues as Elements of Information Systems. Institute of Urban and Regional Development Working Paper 131, University of California, Berkeley, CA, 1970. http://www.cc.gatech.edu/~ellendo/rittel/rittel-issues.pdf. Acesso em 29/09/2004.

8. Lee, J. *Design Rationale Systems: Understanding the Issues*, IEEE Expert Volume 12, No. 13, p. 78-85, 1997.

9. Medeiros, A. P. (2006). Kuaba: Uma Abordagem para Representação de Design Rationale para o Reuso de Designs baseados em Modelo. Tese de Doutorado, Departamento de Informática, PUC-Rio.

10. Nunes, T. R. & Medeiros, A. P. (2009). KSE - ferramenta de apoio à captura e representação semi-automática de design rationale em Engenharia de Software. *Sessão de Ferramentas de XXIII Simpósio Brasileiro de Engenharia de Software* (SBES). Fortaleza, CE.

11. Nunes, T.R. (2010). Ferramenta de Apoio à Captura e Representação de Conhecimento em Projetos de Software usando a Abordagem Kuaba. Relatório Técnico – Universidade Candido Mendes (UCAM-Campos).

12. OMG (2005). Unified Modeling Language Specification. http://www.omg.org/spec/UML/. Acesso em 10/07/2016.

13. OWL (2004). OWL Web Ontology Language Guide. https://www.w3.org/TR/owl-guide/. Acesso em 10/07/2016.

14. Sommerville, I. (2008). *Software Engineering*. 9º Edition, Pearson.

15. RDF (2004). Resource Description Framework. https://www.w3.org/TR/2004/REC-rdf-schema-20040210/. Acesso em 19/07/2016.