Utilização de Sistemas Críticos nas Atividades de Engenharia de Domínio e de Aplicações

Hamilton L. R. Oliveira[1] Cleber R. P. Rocha[1] Kleder M. Gonçalves[1]

Cleidson R. B. De Souza[1, 2]
{crbs@ufpa.br}

[1]Departamento de Informática
Universidade Federal do Pará
Belém, Pará, Brazil

[2]Department of Information and Computer Science
University of California, Irvine
Irvine, California, USA

Sumário

O objetivo deste trabalho é descrever o ambiente *ABCDE-Feature*. Este ambiente utiliza três sistemas críticos para apoiar as atividades de engenharia de domínio e engenharia de aplicações. Desta forma, ele permite a construção de modelos de domínio (diagramas de *features*) e modelos de implementação (diagramas de classes). A integração entre os modelos é feita através de críticas ao modelo de classes que são definidas com base nas características identificadas no diagrama de *features*. Desta forma, os benefícios do uso de críticas são unidos às vantagens da utilização da técnica de engenharia de domínio no processo de desenvolvimento de software.
Palavras-chave: Engenharia de domínio, análise de domínio, diagramas de *features*, sistemas críticos, ambientes de projeto orientados a domínio.

Abstract

This paper presents an environment called ABCDE-Feature. It uses three different critiquing systems to support the creation of feature and class diagrams, which are used within domain and application engineering. There are some mappings between these diagrams that are supported by the environment, providing their integration. In other words, constraints defined in the feature diagram are used as critics in the class diagram, hence the environment reminds designers about characteristics identified during domain engineering that could be forgotten during application engineering.
***Keywords**: Domain engineering, application engineering, critiquing systems, domain-oriented design environments.*

1 Introdução

Conhecimento sobre o domínio da aplicação é fundamental em qualquer processo de desenvolvimento de software. Por exemplo, Curtis *et al.*[4] identificou que uma das principais causas de falhas durante o desenvolvimento de software é a falta de conhecimento do domínio da aplicação. Visando resolver este problema, a área chamada *engenharia de domínios* tem por objetivo fornecer mecanismos para que o desenvolvedor possa compreender os conceitos do domínio da aplicação e representá-los adequadamente. Isto é, os conceitos são

representados genericamente de modo que possam ser (re)utilizados no processo de desenvolvimento de software.

Entretanto, apenas o conhecimento sobre o domínio da aplicação não garante que esta seja desenvolvida adequadamente, uma vez que, durante o projeto, um desenvolvedor pode cometer diversos erros. Para evitar este problema, o uso de sistemas críticos pode auxiliar a detecção de construções de projeto inadequadas e possíveis erros. Um sistema baseado em críticas, doravante chamado sistema crítico, é um software que monitora as ações do usuário e ativa um sinal quando qualquer uma dessas ações viola uma regra definida (uma crítica) [6].

O objetivo deste trabalho é descrever o ambiente *ABCDE-Feature* que apóia as atividades de engenharia de domínio (ED) e de engenharia de aplicações (EA) através da utilização de sistemas críticos. Desta forma, as vantagens das duas abordagens (sistemas críticos e engenharia de domínio) são combinadas visando minimizar os erros cometidos durante o desenvolvimento de software. Este ambiente permite a construção de diagramas de *features*, utilizados na ED, e de diagramas de classes segundo a notação UML, utilizados na EA, além de permitir a conexão entre eles. Assim, o usuário pode criar diagramas de *features* para domínios nos quais ele tenha intenção de desenvolver produtos de software. Durante a criação destes modelos um sistema crítico é utilizado de modo a identificar construções inadequadas. Posteriormente, baseado no diagrama de *features* desenvolvido, o usuário pode desenvolver aplicações para este domínio criando diagramas de classes para aplicações específicas. Neste caso, um segundo sistema crítico é utilizado visando identificar construções que possam resultar em futuros problemas de manutenção e reutilização, isto é, aspectos do sistema que devem ser modificados para torná-los mais flexíveis e reutilizáveis[23]. Além disso, este sistema crítico detecta potenciais erros no mapeamento entre o diagrama de *features* e o diagrama de classes auxiliando os projetistas a desenvolver aplicações que adequadas, ou seja, aplicações que não violam as restrições do domínio.

Desta forma, este trabalho apresenta duas contribuições distintas: (i) o desenvolvimento de um sistema crítico para modelos de domínio (*features)*, assim como a apresentação de críticas para estes modelos, e (ii) o desenvolvimento de uma ferramenta de apoio a atividade de engenharia de aplicações, que utiliza os modelos de domínio desenvolvidos para efetuar críticas durante o desenvolvimento de aplicações para o domínio correspondente.

O restante do artigo está organizado como segue. Na seção 2 são apresentados os conceitos de engenharia de domínio e diagramas de *features.* A seção 3 apresenta brevemente o conceito de críticas, sistemas críticos, os mecanismos de atuações destes e sua importância no desenvolvimento de software. A seção 4 descreve o *ABCDE-Feature*, seu funcionamento e detalhes de implementação. Posteriormente, trabalhos relacionados e comparações com outras ferramentas são apresentados. Finalmente, a seção 6 conclui as idéias abordadas e apresenta sugestões para trabalhos futuros.

2 Engenharia de Domínio e Engenharia de Aplicações

O objetivo da engenharia de domínio é identificar, construir, catalogar e disseminar um conjunto de componentes de software que possuam aplicabilidade em aplicações existentes e futuras em um domínio em particular[19]. A ED inclui um conjunto de métodos e procedimentos para reutilização de software que tem sido desenvolvida desde a década de 80.

Um domínio pode possuir diferentes significados. Neste artigo, um domínio pode ser definido como uma área de conhecimento caracterizada por um conjunto de problemas com especificações funcionais, técnicas e operacionais semelhantes[12]. Ele geralmente apresenta conceitos bem definidos e coerentes, a partir dos quais uma linha de produtos de software que compartilha características poderá ser gerada. Desta forma, é importante que o desenvolvedor

tenha a seu dispor técnicas que o auxiliem a compreender estes conceitos e representar o conhecimento adquirido de forma organizada e de fácil acesso. A ED[14] é uma técnica utilizada para organizar e disponibilizar conhecimento do domínio que possa ser útil ao desenvolvedor não familiarizado com o problema.

Conhecimento sobre o domínio das aplicações é essencial no processo de desenvolvimento de software. De acordo com [4], a falta deste conhecimento é um dos principais problemas enfrentados pelos engenheiros de software. Este problema faz com que os requisitos de uma aplicação sejam mal compreendidos tornando o processo de desenvolvimento mais propenso a erros.

Desta forma, a engenharia de domínio permite criar uma infra-estrutura:

(i) *de conhecimento*, através da qual os usuários desta infra-estrutura podem aprender sobre o domínio, aumentando-se assim, as chances de elaboração de um projeto melhor estruturado, corretamente modelado e menos propenso a erros; e

(ii) *de reutilização*, pois acelera o desenvolvimento de futuras aplicações semelhantes, além de aumentar a confiabilidade destas aplicações através do uso de componentes[1], classes, *frameworks* ou outras estruturas computacionais previamente testadas e portanto, mais confiáveis.

2.1 Etapas da Engenharia de Domínio

O processo de engenharia de domínio possui três etapas: (a) a análise de domínio, (b) o projeto de domínio e (c) a implementação de domínio. Destas etapas, a mais importante para este trabalho é a análise de domínio, onde os diagramas de *features* são construídos. Esta etapa é apresentada em detalhes a seguir.

Na análise de domínio, operações e objetos comuns, características, padrões e procedimentos de sistemas similares em um domínio são identificados e generalizados. A partir disso, um modelo é definido para servir como uma fonte unificada de definições para referência, um repositório de conhecimento compartilhado para a implementação de componentes reutilizáveis[15].

A análise de domínio deve considerar as especificações de uma família de sistemas (aplicações que compartilham características comuns) assim como antecipar as possíveis mudanças que possam ocorrer nestas especificações. Isto deve ser feito de tal forma que os modelos criados possam “gerar” sistemas alvos, e ser capazes de evoluir suas especificações, além de acrescentar novas informações que possam surgir.

Esta etapa provê benefícios como: entendimento dos conceitos pelos envolvidos no projeto; criação de um vocabulário comum entre os desenvolvedores de software, especialistas no assunto e usuários finais; melhor planejamento e conseqüente redução de custos; facilidade de modificações; identificação de características semelhantes entre produtos diferentes do mesmo domínio e, finalmente, reutilização[8]. Em geral, a análise de domínio utiliza diagramas de *features* para representar características do domínio, assim como suas similaridades e diferenças. Estes diagramas serão discutidos em detalhes na seção 2.3.

Após a etapa de análise do domínio, o projeto (de domínio) utiliza os artefatos que foram criados para construir uma arquitetura de software que implemente soluções para os requisitos comuns do domínio. Finalmente, a implementação do domínio transforma as oportunidades de reutilização e soluções do projeto para um modelo de implementação, que inclui serviços como: identificação, reengenharia e/ou construção, e manutenção de componentes reutilizáveis que suportem estes requisitos e soluções do projeto.

2.2 Engenharia de Aplicações

Aliado à engenharia de domínio (ED) encontra-se a engenharia de aplicações (EA) que utiliza os produtos gerados pela ED para a construção de aplicações específicas. De um modo geral, pode-se dizer que uma aplicação é criada "fatiando-se" o modelo de domínio de acordo com os requisitos individuais desta aplicação. Isto é, os requisitos específicos da aplicação são identificados e as *features* que implementam estes conceitos são selecionadas para servirem como infra-estrutura para a aplicação.

Durante a EA, escolhe-se os componentes necessários à aplicação num alto nível de abstração, através do modelo do domínio, descendo gradualmente em níveis de abstração até atingir os componentes implementados ou semi-desenvolvidos do domínio[16].

2.3 O Diagrama de *Features*

Conforme apresentado na seção anterior, uma das tarefas mais importantes da análise de domínio envolve a construção de diagramas de *features* que são diagramas que representam as características do domínio, suas similaridades e diferenças. Estes diagramas apresentam as características (*features*) similares às aplicações de um domínio, assim como outras características que podem opcionalmente ser incorporadas, os limites ou "fronteiras" do domínio e o relacionamento deste com os domínios "vizinhos" identificados. De um modo geral, *features* são abstrações funcionais a serem empacotadas, implementadas, testadas e mantidas, na forma de classes ou de componentes reutilizáveis.

Os diagramas de *features* devem ser suficientemente genéricos e abrangentes para que sejam válidos para outras aplicações pertencentes ao mesmo domínio. Por isso, eles devem ser validados por especialistas no domínio e, idealmente, podem ser testados utilizando-se aplicações que fazem parte do domínio, mas que não foram utilizadas no processo de definição do mesmo.

Estes diagramas também mostram a decomposição arquitetural das *features*, suas definições, tipos e regras de composição, indicando quais são opcionais, quais são mutuamente exclusivas ou necessitam de outros pré-requisitos. Para representar estas informações, notações específicas são utilizadas na construção destes diagramas. A próxima seção descreve a notação utilizada neste trabalho, enquanto que um exemplo de um diagrama de *features* criado utilizando-se o *ABCDE-Feature* é apresentado na seção 4.

2.4 Notação para Diagramas de *Feature*s

Na literatura existem diversas notações para representar os modelos de *features* tais como FODA[14], FODACom[25], [17], etc. etc. Neste trabalho, a notação adotada foi a proposta por Miler[16]. Esta notação utiliza um conjunto de características herdadas de métodos consagrados, como exclusividade e opcionalidade, além de outras adicionais como restrição e expansibilidade.

Para possibilitar a modelagem de um domínio esta notação utiliza seis diferentes tipos de *features* descritas a seguir:

1) Essenciais: Correspondem à características fundamentais do modelo. São *features* intimamente ligadas à essência do domínio. Elas descrevem características que representam as funcionalidades/conceitos do modelo.

2) Organizacionais: São características do modelo que têm apenas o intuito de facilitar o entendimento ou organizar o domínio. Não possuem ligações concretas com o uso real do domínio.

3) Entidade: São os atores do modelo, ou seja, entidades do mundo real que atuam sobre o domínio. Podem, por exemplo, expor a necessidade de uma interface ou procedimentos de controle.

4) Externas: São *features* que pertencem a outros domínios. Elas podem ou não ser definidas pelo modelo, pois mostram a fronteira do domínio e como ele se comporta.

5) Não definidas: Correspondem à características previamente identificadas em um domínio, porém ainda não definidas através de outros modelos.

6) Adicionais: Características adicionais, mas importantes para o entendimento do domínio.

Features são utilizadas para representar os conceitos mais importantes envolvidos em um domínio. Relacionamentos são elementos que possibilitam expressar a forma como estas *features* interagem. Os relacionamentos propostos por [16] são os seguintes:

1) Composição: Neste relacionamento, uma feature é composta de várias outras. É uma relação onde uma feature é parte fundamental de outra, de forma que uma não existe sem a outra.

2) Agregação: É um relacionamento no qual uma *feature* representa o todo, e as outras representam as partes. É similar a composição sem a relação de dependência entre seus membros.

3) Herança: Neste relacionamento, as *features* filhas são subtipos da *feature* mãe. Neste caso, as *features* filhas herdam as características de seus antecessores.

4) Associação: Ligação simples entre duas características. Denota algum tipo de relacionamento entre seus membros. Pode ser nomeada, indicando um tipo específico de ligação.

Os relacionamentos descritos anteriormente são intencionalmente similares a relacionamentos da notação UML[16]. Além destes, outros relacionamentos provenientes de outros métodos de engenharia de domínio são utilizados, tais como:

1) Exclusividade: Relacionamento no qual as *features* filhas não podem ser usadas ao mesmo tempo. Isto pode denotar variações, problemas de incompatibilidade ou impossibilidade de implementação de uma extensão.

2) Opcionalidade: Denota uma característica não mandatória do domínio.

3) Restrição: Problemas ou necessidade de uso conjunto de duas *features*, que não necessariamente possuem relacionamento direto.

4) Expansibilidade: Permite ao diagrama a flexibilidade de navegação em diferentes níveis de detalhamento. Expressa a necessidade de um refinamento da característica em questão.

Um exemplo de um diagramas de *features*, criado utilizando-se o *ABCDE-Feature*, é apresentado na seção 4.

3 Sistemas Críticos

Um sistema crítico é um software que monitora as ações do usuário e ativa um sinal quando qualquer uma dessas ações viola uma regra definida (uma crítica) [13][23]. Por exemplo, no desenvolvimento de software, uma crítica poderia ser formulada com a seguinte definição: "*Uma classe não deve ter atributos públicos*". Esta crítica seria disparada quando um usuário criasse uma classe com um atributo público. Neste momento, o sistema crítico deveria então apresentar a justificativa para a crítica ter sido acionada. Isto pode ser feito através da apresentação de argumentos que auxiliem o usuário a entender melhor o problema e contribuam para sua tomada de decisão. Desta forma, o objetivo dos sistemas críticos é auxiliar os usuários durante a construção do projeto através da apresentação de *feedback* no contexto da tomada de decisão[6].

Os sistemas críticos são adequados a domínios de problema com as seguintes características[13][6]: (a) o conhecimento sobre o domínio está incompleto e evoluindo; (b) os requisitos do problema podem ser especificados somente parcialmente; e (c) o conhecimento de projeto necessário está distribuído entre muitos membros do projeto. Desta forma, pode-se concluir facilmente que sistemas críticos podem ser utilizados para auxiliar a atividade de desenvolvimento de software. De fato, pode-se observar na literatura alguns trabalhos nesta direção (veja seção sobre trabalhos relacionados).

Uma importante característica dos sistemas críticos é que a expansão ou modificação do conhecimento por especialistas do domínio pode ser feita de modo relativamente simples. Isto é possível porque, geralmente, as críticas são independentes umas das outras; o que permite que novas críticas possam ser adicionadas sem conflitos ao conjunto de críticas existentes.

Formalmente, críticas de computador são compostas de conjuntos de regras ou procedimentos para avaliar diferentes aspectos de um produto. Elas podem ser baseadas em princípios físicos, axiomas, ou na experiência de especialistas (conhecimento heurístico). Seu objetivo é fazer com que o desenvolvedor reflita sobre uma solução implementada, verificando se ela é a mais adequada ou se deve ser modificada. As críticas podem auxiliar na detecção de: possíveis erros no projeto, soluções problemáticas, inconsistências entre o projeto e suas especificações e violações de padrões. Isto pode otimizar o processo de desenvolvimento, não apenas eliminando erros, mas também auxiliando o projetista a desenvolver melhores soluções para o problema.

É importante observar que uma crítica é disparada em situações propensas a erros, alertando o desenvolvedor sobre possíveis falhas de projeto. Em outras palavras, uma crítica também pode ser considerada uma recomendação ao usuário, portanto ele não precisa necessariamente concordar com a crítica, mas estará ciente das implicações de sua atitude. De acordo com [13], as críticas podem até mesmo ser usadas para manter uma base de conhecimento com soluções alternativas e conflitantes para um determinado problema. Por exemplo, em um projeto orientado a objetos onde existe uma associação entre as classes *A* e *B* com multiplicidade "*muitos-para-muitos*", existem dois conselhos possíveis: (i) a adição de um atributo de ligação para modelar as propriedades da associação [20]; ou (ii) a adição de uma nova classe, por exemplo, *C*, a qual possui duas associações: uma com a classe *A* e outra com *B*[5]. Neste caso, as propriedades são adicionadas como atributos da nova classe. Então, esta crítica pode ser implementada com uma cláusula de condição que identifica associações *muitos-para-muitos*, e, em sua argumentação, os dois conselhos possíveis são descritos e avaliados. Cabe ao usuário escolher qual das duas soluções é mais adequada para seu trabalho.

Finalmente, um sistema crítico deve compreender o domínio do problema a fim de criticar as características inerentes a todos os produtos daquele domínio[6][18]. Além disso, ele deve possuir estratégias de ativação que determinem quais aspectos de um projeto de usuário deve ser criticado, quando e como isto deve ser realizado. Isto é necessário porque, segundo [6], o disparo de críticas em momentos inoportunos pode tornar-se um incômodo ao usuário. É também desejável que o sistema crítico permita ao usuário adicionar ou modificar críticas, decidir sobre a forma de atuação das mesmas e até mesmo desabilitar as que ele considerar inadequadas.

As diversas características descritas anteriormente são implementadas no ambiente *ABCDE-Feature*, visto que este reutiliza o sistema crítico para diagramas de projeto orientados a objetos *ABCDE-Critic*[23], que implementa tais funcionalidades. O *ABCDE-Feature*, assim como o *ABCDE-Critic* serão descritos na próxima seção.

4 O Ambiente *ABCDE-Feature*

Esta seção descreve o ambiente *ABCDE-Feature*. A seção 4.1 descreve a motivação para o desenvolvimento e utilização desta ferramenta, enquanto que a seção 4.2 descreve a arquitetura do *ABCDE-Feature*. O sistema crítico para o diagrama de *features* é apresentado assim como críticas para tais diagramas, bem como o sistema crítico para o diagramas de classes e exemplos de críticas geradas a partir do domínio.

4.1 Motivação

Segundo [4] a falta de conhecimento sobre o domínio é uma das principais causas de falhas no desenvolvimento de software, pois, leva a uma incorreta modelagem da aplicação a ser desenvolvida, tornando o projeto mais propenso a erros. Este problema pode ser resolvido utilizando-se diversas abordagens, entre elas a construção de modelos do domínio, conforme discutido na seção 2. A construção destes modelos pode ser auxiliada com a utilização de sistemas críticos que identificam potenciais problemas no diagrama (seção 3).

Além disso, o mapeamento dos modelos de domínio (neste caso, os diagramas de *features*) para modelos de implementação (diagramas de classes) também é uma tarefa difícil e propensa a falhas. Novamente, sistemas críticos podem ser utilizados para auxiliar este processo: tais sistemas avaliam se os modelos de implementação desenvolvidos refletem adequadamente os modelos de domínio nos quais eles se baseiam. Isto é possível porque a informação coletada através dos modelos de domínios é utilizada para habilitar críticas que avaliam os diagrama de classes construídos.

Neste trabalho, três diferentes sistemas críticos foram utilizados:

(i) o primeiro avalia os diagramas de *features* avaliando se estes diagramas se adequam à notação utilziada, (ii) o segundo sistema crítico avalia o diagrama de classes de acordo com heurísticas de projeto orientado a objetos, e (iii) finalmente o último verifica se o diagrama de classe atende às restriçoes impostas pelo diagrama de *features*. Em resumo, esta abordagem fornece apoio ao desenvolvimento de diagramas de *features* e classes, assim como lembra os desenvolvedores de restrições identificadas durante a ED que poderiam ser esquecidas durante a EA. Estas idéias foram implementadas no ambiente *ABCDE-Feature* descrito a seguir.

4.2 O Ambiente *ABCDE-Feature*

O *ABCDE-Feature* é um ambiente desenvolvido em Java composto de aproximadamente 40 classes. Ele possui duas aplicações principais: o *FeatureEditor* e o *ClassEditor*. Estas aplicações foram desenvolvidas a partir do sistema crítico para modelos de

objetos chamado *ABCDE-Critic*[23]. Este, por sua vez é uma instância do framework para editores de diagramas cooperativo ABCDE[24](*Anotation Based Cooperative Diagram Editor*). O *ABCDE-Critic* utiliza o motor de inferência de primeira ordem JEOPS[9]. A arquitetura do *ABCDE-Feature* é apresentada na Figura 1.

Inicialmente, os elementos da Figura 1 são brevemente apresentados. A seguir, os dois elementos do *ABCDE-Feature* (*FeatureEditor* e *ClassEditor*), são descritos detalhadamente.

ABCDE é um *framework* orientado a objetos para o desenvolvimento de editores de diagramas com apoio a anotações. Um dos pontos adaptáveis[26] do ABCDE permite a construção de editores de diagramas para diferentes notações. Esta característica do framework foi herdada pelo ambiente *ABCDE-Critic* o que possibilitou a criação do editor de diagramas para modelos de *features* chamado *FeatureEditor*.

ABCDE-Critic é um sistema crítico para diagramas de classes UML. As críticas no *ABCDE-Critic* podem estar em três estados diferentes: (i) ativas, que monitoram as ações do usuário e respondem instantaneamente a elas, (ii) passivas, que são invocadas pelo usuário para avaliar o produto parcial desenvolvido, ou (iii) desabilitadas, quando não são executadas. As críticas podem ser definidas pelo próprio usuário dentro do ambiente.

JEOPS (Java Embedded Object Production System) é um sistema de produção que permite desenvolver aplicações inteligentes utilizando-se a linguagem Java, através de uma estratégia de programação declarativa. Este sistema possibilita a integração entre regras de produção e a linguagem Java, o que permite o desenvolvimento de aplicações inteligentes nesta linguagem.

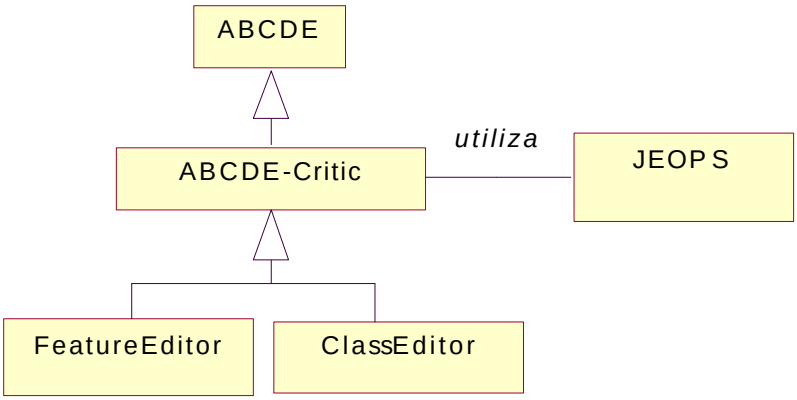

Figura 1: Componentes do ambiente *ABCDE-Feature*

O *FeatureEditor*

Conforme discutido anteriormente, o *FeatureEditor* foi implementado a partir do *ABCDE-Critic*. Este procedimento possibilitou que fossem herdadas todas as funcionalidades implementadas pelo *ABCDE-Critic*, inclusive o sistema crítico. Desta forma, a primeira contribuição deste trabalho é o desenvolvimento de um sistema crítico para modelos de *features*. O *FeatureEditor* utiliza diagramas de *features* para representar os conceitos do domínio segundo a notação proposta por [16] e descrita na seção 2.3.

Em resumo, o objetivo do *FeatureEditor* é possibilitar a criação e manipulação de diagramas de *features* estendidos, permitindo que as *features* de um domínio possam ser identificadas, assim como os relacionamentos entre elas. A Figura 2 mostra a tela do *FeatureEditor* exibindo parte do diagrama de *features* do domínio de sistemas de telefonia extraído de [16].

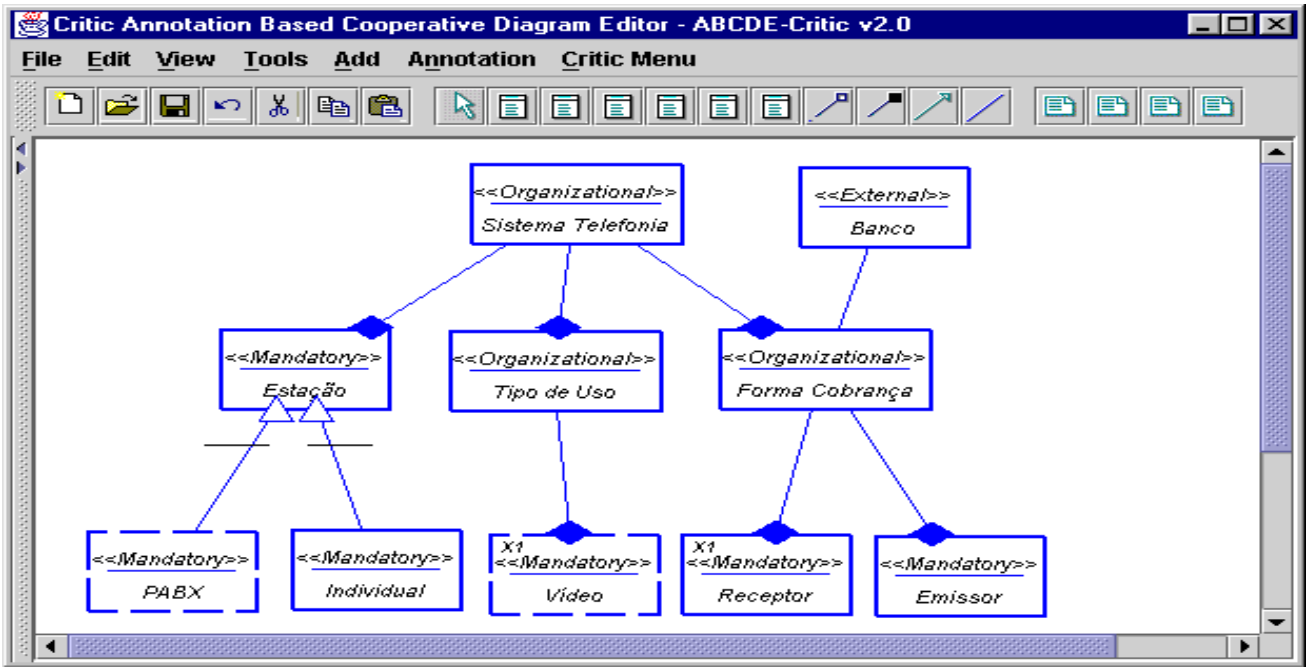

Figura 2: Tela do *FeatureEditor* com o diagrama do Sistema de Telefonia

A seguir um exemplo de crítica para modelos de *features*, assim como sua justificativa, são apresentados:

Features Externas não devem ser Refinadas

Features externas são aquelas pertencem a outros domínios[16]. Elas mostram a fronteira do domínio e seu comportamento. Neste caso, como estas features não pertencem ao domínio do problema, idealmente, elas não precisam ser refinadas, visto que não são realmente importantes neste domínio. Na verdade se estas críticas são refinadas elas estão adicionando uma complexidade desnecessária ao modelo, além de torná-lo mais propenso a falhas. Por exemplo, um projetista pode inadvertidamente conectar uma feature essencial do modelo a uma sub-feature de uma feature externa.

A definição da crítica acima utilizando-se a sintaxe do JEOPS é apresentada abaixo:

```
rule ExternalFeature{
declarations
        feature.critic.CriticFeature cf;
        feature.model.External _external;
        feature.model.Feature _feature;
        feature.model.Relationship _relationship;
localdecl
        java.util.HashMap hm=new java.util.HashMap();
preconditions
        _external.containsConector();
        !_relationship.isAssociation();
        (_relationship.isEndNode(_external)||
          _relationship.isStartNode(_external));
        (_relationship.isEndNode(_feature)||
          _relationship.isStartNode(_feature));
actions
        hm.put("ExternalFeature",_external);
        hm.put("RefinedFeature",_feature);
        hm.put("ExternalRelationship",_relationship);
        cf.fire(hm);
}
```

No momento, o ambiente *ABCDE-Feature* está sendo "semeado"[7] com novas críticas a serem utilizadas no diagrama de *features*. É importante notar que o *ABCDE-Feature* também permite que os usuários adicionem suas próprias críticas de acrodo com a sintaxe do JEOPS. Isto permite que especialistas do domínio também possam codificar parte de seu conhecimento na forma de críticas. Por exemplo, no domínio de sistemas de a telefonia, pode-se definir que uma Estação PABX deve possuir no máximo 5 instâncias. Esta informação não pode ser expressa no diagrama de *features*, no entanto pode ser definida no JEOPS e portanto em nosso ambiente.

O *ClassEditor*

O *ClassEditor* é o componente mais importante do ambiente *ABCDE-Feature*. Ele implementa um editor de diagramas de classes segundo a notação UML integrado a dois sistemas críticos. Ele auxilia os usuários durante a atividade de engenharia de aplicações, isto é, ele permite a construção de diagramas de classes que modelam as aplicações a serem desenvolvidas para o domínio previamente modelado usando o *FeatureEditor*.

De maneira similar ao *FeatureEditor*, ele foi desenvolvido a partir do *ABCDE-Critic*, portanto, as críticas previamente implementadas são reutilizadas pela ferramenta. Isto permite o desenvolvimento de modelos orientados a objetos flexíveis e reutilizáveis, pois atendem às críticas embutidas no ambiente e descritas em [10],[23].

Além disso, o *ClassEditor* também utiliza o modelo de domínio (através dos diagramas de *features* desenvolvidos) para identificar críticas no diagrama de classes. Isto é possível, uma vez que o modelo de *features* cria restrições às implementações deste modelo. Ou ainda, o modelo de *features* sugere recomendações no modelo de classes a ser desenvolvido. Estas restrições e sugestões foram mapeadas como críticas no *ClassEditor*. Exemplos de críticas e suas justificativas são apresentados no decorrer do texto.

As classes que descrevem a aplicação em construção são conectadas às *features* do modelo do domínio do qual elas se originaram. Esta conexão pode ser feita de várias formas diferentes, portanto a cardinalidade da conexão é "*muitos-para-muitos*", isto é, uma *feature* pode ser implementada como várias classes, assim como várias *features* podem ser mapeadas para uma única classe. Este mecanismo de "conexão" entre *features* e classes é chamado *traceability*[2] e foi implementado como uma classe que possui referências para as respectivas *features*/classes que estão conectadas. Isto permite uma navegação bidirecional entre os modelos: a partir de uma classe, pode-se identificar a(s) feature(s) que ela implementa e, dada uma *feature*, pode-se determinar a(s) classe(s) que a implementa(m). O usuário também pode escolher o modelo de domínio que ele pretende utilizar. Opcionalmente, este modelo é apresentado durante a construção do diagrama de classes através da janela chamada "*Show Feature Diagram*".

Figura 3 apresenta o ambiente *ClassEditor* no momento em que uma crítica derivada a partir do modelo de domínio é disparada no diagrama de classes. Neste exemplo, conforme pode ser observado, as *features* do modelo do domínio foram originalmente criadas com o relacionamento de herança (*pai-filha)* entre elas[1]. No entanto, as classes que implementam estas *features* possuem o relacionamento de agregação entre elas. Este situação ocasionou o disparo da crítica, visando manter a integridade dos conceitos e relacionamentos previamente definidos no diagrama de *features.* Neste momento, o usuário pode aceitar a sugestão do sistema e efetuar as modificações nos elementos indicados no diagrama de classes. Estes

[1] Apenas com o propósito de simplificar o exemplo apresentado, o mapeamento de *features* para classes foi feito diretamente, isto é, uma *feature* é implementada como uma classe e as classes que implementam as *features* foram identificadas com o mesmo nome. Conforme discutido no texto, este mapeamento pode ser implementado de diversas maneiras diferentes.

elementos são os responsáveis pelo disparo da crítica. O sistema permite ainda que o usuário desabilite a crítica, significando que está consciente do alerta enviado pelo ambiente mas que prefere manter as divergências entre os diagramas.

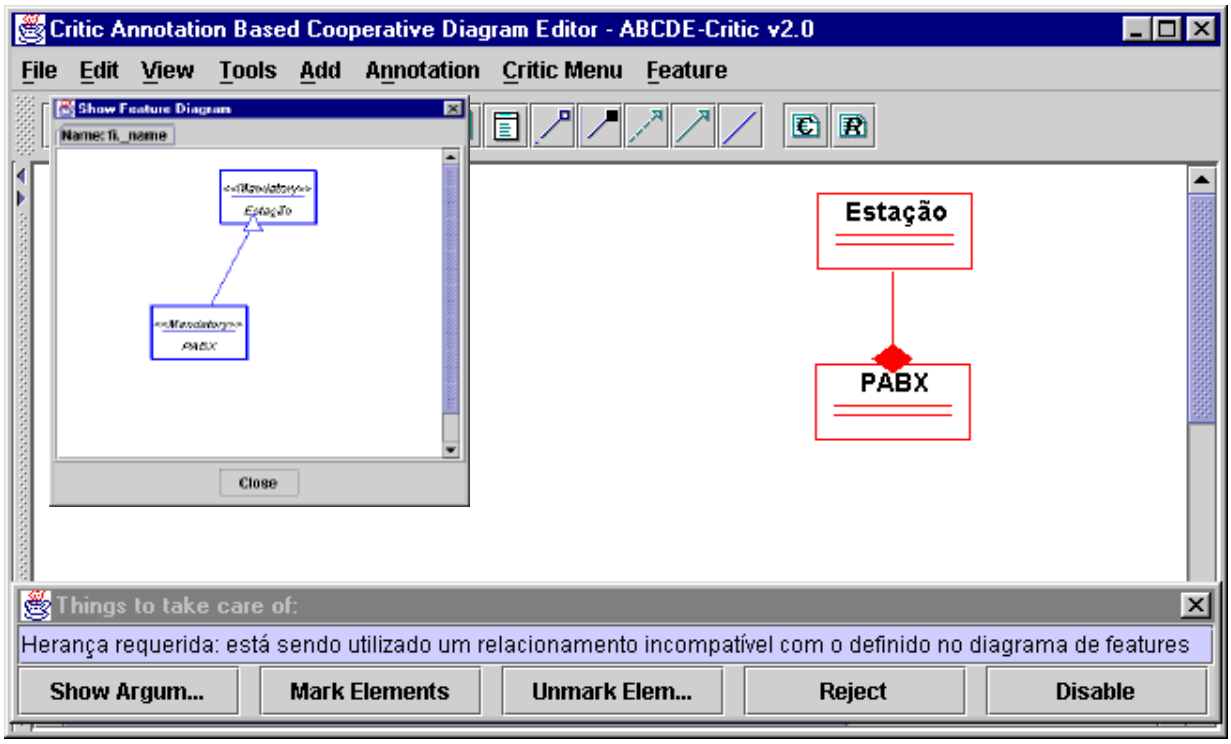

Figura 3: O ambiente *ABCDE-Feature* no momento em que uma crítica é disparada.

A seguir, dois outros exemplos de críticas são apresentados:

Features Essenciais

Uma outra crítica importante refere-se às features essenciais no modelo do domínio. Estas features modelam características fundamentais do domínio. Elas estão intimamente ligadas à essência do domínio. Portanto, features essenciais devem ser implementadas durante a implementação do domínio, e consequentemente serão reutilizadas durante a engenharia das aplicações. Isto corresponde a afirmar que estas features devem ser mapeadas para classes no modelo de classes das aplicações que estão sendo desenvolvidas neste domínio. Desta forma, a crítica verifica se as features consideradas essenciais no modelo do domínio são implementadas como classes no modelo de classes. Este é o típico exemplo de crítica que deve ser entendida como uma recomendação, visto que dependendo da aplicação a ser modelada, nem todas as features essenciais serão importantes. Ainda assim, os autores acreditam que esta crítica pode ser útil durante o processo de engenharia de aplicações.

Restrições entre Features

Miler[16] define o conceito de restrição entre features (seção 2.4) que significa que (i) podem ocorrer problemas quando duas features são utilizadas em conjunto ou (ii) que duas features devem obrigatoriamente ser utilizadas em conjunto. Este tipo de restrição pode ocorrer entre features não necessariamente conectadas por outros relacionamentos.

Neste caso, pode-se observar que restrições podem ser facilmente utilizadas para a geração de críticas. Por exemplo, se duas features devem ser utilizadas em conjunto, então ambas devem possuir um mapeamento no diagrama de classes correspondente. Ou, de maneira oposta, se tais features não podem ser utilizadas em conjunto, deve-se verificar o diagrama de classes para evitar a violação desta restrição.

Entretanto, a notação de [16] não prevê a separação entre estes dois casos. Desta forma, os autores optaram por criar dois subtipos de restrições chamadas de restrições de inclusão e restrições de exclusão. A primeira prevê o uso conjunto de features, enquanto que a segunda indica que tal condição não pode ocorrer. Desta forma, permitiu-se o aumento do poder computacional do ambiente ABCDE-Feature (duas novas críticas foram adicionadas) sem aumentar demasiadamente a complexidade da notação para os engenheiros de domínio.

5 Trabalhos Relacionados

Na literatura existem diversas propostas para apoiar a engenharia de domínio. Entretanto, a maioria deles se refere a descrição de métodos e notações com o objetivo de auxiliar esta atividade. Além disso, alguns poucos sistemas críticos para apoio ao desenvolvimento de software são encontrados na literatura. Ambas as abordagens serão comparadas nesta seção.

5.1 Engenharia de Domínio

[11] propôs um protótipo de um ambiente cujo objetivo é apoiar a modelagem de domínios independente de domínio de aplicação. O produto final do processo de modelagem é a geração automática da especificação de um sistema-alvo devidamente contido no respectivo domínio. Este ambiente realiza a verificação do modelo do domínio através do *Domain Model Consistency Checker*(DMCC), que consiste de um conjunto de *scripts*, em uma linguagem de consulta chamada *Troll/USE*. Ele vasculha os diversos diagramas em busca de inconsistências nos relacionamentos existentes entre os mesmos. Essa verificação somente pode ser realizada após a criação de todos os diagramas integrantes da metodologia de domínio utilizada.

O *ABCDE-Feature* monitora as atividades dos desenvolvedores disparando críticas no exato momento em que erros na formalização do conhecimento extraído do domínio são cometidos. Isto evita que mensagens de erro atrasadas pareçam fora de contexto, não prevenindo adequadamente o usuário[6].

O Odyssey[2][3] é um ambiente de reuso de software baseado em modelos de domínio. Sua principal contribuição é relacionar os conceitos de desenvolvimento baseado em componentes e engenharia de domínio. O Odyssey concentra a maior parte de seus esforços na verificação de inconsistências na fase de engenharia de aplicações. Isto é, pouca ênfase foi dada para verificação de inconsistências durante a engenharia do domínio (construção dos diagramas de *features*). Diferentemente, o *ABCDE-Feature* suporta a verificação de consistência tanto na engenharia de domínio quanto na engenharia de aplicações.

Finalmente, quando o ambiente Odyssey detecta incompatibilidades, cabe ao desenvolvedor intervir para que as correções sejam efetuadas. O *ABCDE-Feature* oferece um recurso adicional, que são as chamadas *alternativas*[10]. Este recurso foi herdado do ambiente *ABCDE-Critic* e permite que, associada a uma crítica, seja definida uma ou mais alternativas para o problema que esta indica. Alternativas correspondem a soluções para o problema. Assim, o ambiente *ABCDE-Feature* permite a visualização das soluções no contexto original permitindo uma avaliação mais adequada das soluções. Se o usuário concordar com a crítica ele pode simplesmente aceitar a indicação do sistema crítico que se encarregará de modificar automaticamente o diagrama.

5.2 Sistemas Críticos

Na literatura existem poucos sistemas críticos para apoiar atividades de desenvolvimento de software. Por exemplo, [21] descrevem Argo um sistema crítico com diversas funcionalidades baseadas em teorias cognitivas para apoiar a construção de modelos de arquitetura de software. Posteriormente, em [22] estes autores descrevem a ferramenta ARGO/UML para modelagem orientada a objetos. Argo/UML suporta a edição de diagramas utilizados na notação UML, além de permitir a geração de código Java a partir do diagrama de classes e identificar erros comuns cometidos pelos projetistas. Neste caso, as críticas para os sistemas críticos são baseadas em heurísticas de modelagem orientada a objetos, assim como de especificações da semântica da UML.

Além disso, [23] descrevem o ambiente *ABCDE-Critic* que utiliza um sistema crítico para verificar diagramas de classes construídos segundo a notação UML. Diferentemente do Argo e Argo/UML, o *ABCDE-Critic* permite que os próprios usuários adicionem críticas ao sistema crítico, pois utiliza o sistema de produção JEOPS. Nenhum destes sistemas permite a construção de modelos de domínio.

6 Conclusões e Trabalhos Futuros

Este artigo apresentou o *ABCDE-Feature*, um ambiente que implementa três diferentes sistemas críticos que visam auxiliar a engenharia de domínio e a engenharia de aplicações. O primeiro sistema crítico apóia a engenharia de domínio através de críticas que identificam erros e potenciais problemas na construção de diagramas de *features*. O segundo sistema crítico utiliza heurísticas para modelagem orientada a objetos herdadas do ambiente *ABCDE-Critic*, enquanto que o terceiro e último, utiliza restrições identificadas no modelo de domínio, como críticas que são aplicadas ao diagrama de classes em construção.

Este ambiente fornece ao usuário os benefícios combinados da técnica de engenharia de domínios e dos sistemas críticos, para a obtenção de melhores soluções de projeto a custo e esforço reduzidos.

Atualmente, existem cerca de sete críticas para o modelo de *features*, vinte críticas baseadas em heurísticas para modelos de objetos e sete críticas definidas a partir dos modelos de domínio. Apesar do pequeno número de críticas implementadas, os autores acreditam que esta abordagem é valiosa no processo de engenharia de domínio e de aplicações. Os autores pretendem adicionar outras críticas visando melhorar a integração entre os diagramas de *features* e de classes.

7 Agradecimentos

Os autores gostariam de agradecer a CAPES (processo número BEX1312/99-5) e a Universidade Federal do Pará (programa PIPES de iniciação científica) pelas bolsas de estudo. Além disso, os autores também gostariam de agradecer aos membros do projeto Odyssey (COPPE-UFRJ) e em especial a Profa. Dra. Cláudia Werner pelo apoio recebido durante o desenvolvimento deste trabalho.

8 Bibliografia

[1] Braga, R., Werner, C. *Desenvolvimento Baseado em Componentes*. XIV Simpósio Brasileiro de Engenharia de Software, Minicurso, Outubro, p. 297-329, João Pessoa, PB, 2000.

[2] Braga, R., Werner, C., Mattoso, M. *A Reuse Infrastructure Based on Domain Models*. In: Proceedings of the 5th International Conference on Computing and Information, Winnipeg, Canada.

[3] Braga, R., Werner, C., Mattoso, M. *Odyssey: A Reuse Enviroment Based on Domain Models*. In Proceedings of the 2nd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'99), Richardson, USA, Março, 1999.

[4] Curtis, B., Krasner, h. and Iscoe, N. "A Field Study of the Software Design Process for Large Systems". *Communications of the ACM* 31(11):1268-1287, Novembro, 1988.

[5] Coad, P. e Yourdon, E. Object Oriented Analysis, Prentice-Hall International, Segunda edição 1991.

[6] Fisher, G., Nakakoji, K. Embedding critics in design enviroments. *The Knowledge Engeneering Review*, 8 (4); 285-307, 1993.

[7] Fischer, G. Seeding, Evolutionary Growth and Reseeding: Constructing, Capturing and Evolving Knowledge in Domain-Oriented Design Environments, *International Journal Automated Software Engineering*, Kluwer Academic Publishers, Dordrecht, Netherlands, Vol. 5, No.4, pp. 447-464, October 1998.

[8] Fraser, S., Leishman, D., Mclellan, R. *Patterns, Teams and Domain Engeneering.* In: Proceedings of the 17th international conference on software engineering on Symposium on software reusability, 1995, Pages 222 – 224.

[9] Figueira Filho, C. Ramalho, G. Jeops – the java Embedded Object Production System. In M. Monard e J. Sichman (eds). Advances in Artificial Intelligence, *Lecture Notes on Artificial Intelligence Series*, vol. 1952, pp 52-61. London: Springer-Verlag, 2000.

[10] Ferreira Jr., J. , Souza, C., Wainer, J. *Um sistema crítico e coletor de design rationale integrados em um ambiente para análise e projeto orientados a objetos.* In: Anais do Concurso de Trabalhos de Iniciação Científica, Sociedade Brasileira de Computação, Curitiba, Paraná, Julho, 2000.

[11] Gomaa, H.; Kerschberg, L.; Sugumaran, V.; Bosch, C.; Tavakoli, I., A prototype domain modeling environment for reusable software architectures, In Proceedings of the Third International Conference on Software Reuse: Advances in Software Reusability, Page(s): 74 – 83, 1994.

[12] Gomaa, H. *An object-Oriented domain analysis and modeling method for software reuse*.. Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, Volume: ii , Page(s): 46 –56, 1992.

[13] Hägglund, S. Introducing Expert Critiquing Systems, *The Knowledge Enginnering Review*, vol. 8, n. 4, pp. 281-284, 1993.

[14] Kang, K., Cohen, S., Hess, J. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Relatório Técnico CMU/SEI-90-TR-21, Software Engineering Institure, Carnegie Mellon University, Pittsburgh, 1990.

[15] Lung, C., Urban, J. *Integration of Domain Analysis and Analogical Approach for Software Reuse.* In: Proceedings of the ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice, Pages 48 – 53, 1993.

[16] Miler Jr., N. *A Engenharia de Aplicações no Contexto da Reutilização Baseada em Modelos de Domínio.* Dissertação de Mestrado. Universidade Federal do Rio de Janeiro, Rio de Janeiro - RJ, 2000.

[17] Morisio, M.; Travassos, G.H.; Stark, M.E. *Extending UML to support domain analysis*, In Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering, Page(s): 321 –324, 2000.

[18] Oliveira, K., Travassos, G., Menezes, C., Rocha, A. *Ambientes de Desenvolvimento de Software Orientados a Domínio*. In: Anais do XIV Simpósio Brasileiro de Engenharia de Software**.** João Pessoa, PB. Outubro, 2000. p.275-290, 2000.

[19] Pressman, R. S. *Software Engineering: A Practitioners Approach*, Fifth edition, McGraw-Hill, 2000.

[20] Rumbaugh, J., Blaha, M., Prelermani, W. *et al.* Object-Oriented Modeling and Design, Prentice Hall International, 1991.

[21] Robbins, J, Hilbert, D.M. e Redmiles, D.F. *Extending Design Environments to Software Architecture Design*, Proceedings of the Knowledge-Based Software Engineering, 1996.

[22] Robbins, J, e Redmiles, D.F, *Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML*, Proceedings of the International Conference on Construction of Software Engineering Tools,1999.

[23] Souza, C.R.B.; Ferreira, J.S., Jr.; Goncalves, K.M.; Wainer, J. *A group critic system for object-oriented analysis and design*, In Proceeding of The Fifteenth IEEE Conference on Automated Software Engineering, 313-316, IEEE Press, 2000.

[24] Souza, C. *Um Framework para Editores de Diagramas Cooperativos baseados em Anotações.* Dissertação de Mestrado. Instituto de Computação – Universidade Estadual de Campinas, Campinas – SP, Outubro, 1998.

[25] Vici, A.D.; Argentieri, N.; Mansour, A.; d'Alessandro, M.; Favaro, J., FODAcom: an experience with domain analysis in the Italian telecom industry **,** In Proceedings of the Fifth International Conference on Software Reuse, Page(s): 166 –175, 1998.

[26] Wolfgang, Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley Publishing. 1995.