Formalização de Heurísticas para o Apoio a Modelagem de Sistemas Orientados a Objetos

André Luis Castro de Freitas *,**
Ana Maria de Alencar Price ***

* Universidade Católica de Pelotas
Escola de Informática
** Escola Técnica Federal de Pelotas
Unidade de Processamento de Dados
afreitas@inf.ufrgs.br

*** Universidade Federal do Rio Grande do Sul
Instituto de Informática
Pós-Graduação em Ciência da Computação
anaprice@inf.ufrgs.br

Resumo

Com o objetivo de auxiliar a formalização do processo de desenvolvimento de *software* orientados a objetos, este trabalho propõe heurísticas que possibilitem tornar mais automáticas as atividades de modelagem e projeto. As heurísticas propostas estão classificadas em grupos representando contribuições para a construção de classes e objetos, e estão ilustradas e demonstradas, em diversos casos, através de exemplos. O desenvolvimento dos exemplos permitiu a comparação entre as diversas heurísticas, mostrando ainda tendências e vantagens na utilização de determinado tipo de comunicação entre classes e objetos. A partir do estudo das heurísticas construiu-se uma ferramenta que visa auxiliar o projetista na definição de um bom projeto.

Palavras-chave: Engenharia de *Software,* Projeto Orientado-Objeto, Heurísticas, Padrões de Projeto.

Abstract

Aiming to formalize the development process of object oriented Software, this work proposes heuristics to enable the process of modeling and design as automatic as possible. These heuristics are classified in groups contributing for: class and object building. The heuristics are explained and demonstrated by examples. The development of these examples allows heuristics comparison, showing advantages and tendencies in using one or another kind of communication between classes and objects. The heuristics studies stimulated to create a tool with purpose of helping designer in definition of good designs.

Keywords: Software Engineering, Object-Oriented Design, Heuristics, Design Patterns.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION

1 Introdução

As metodologias de desenvolvimento e as linguagens de programação evoluíram durante as últimas décadas, com o objetivo de minimizar os problemas de manutenção de *software*. Entretanto estas mesmas metodologias falham no fornecimento de suporte adequado à compreensão dos sistemas envolvidos. Segundo Arango [ARA 93], as metodologias de desenvolvimento de software orientado a objetos, em geral, *"falham na provisão de formalismos adequados para refletir o mapeamento entre o projeto e a implementação"*.

A utilização de ferramentas, automáticas ou manuais, no trabalho dos analistas e projetistas durante o desenvolvimento de software orientado a objetos é de grande importância para ajudar a reduzir a complexidade inerente ao processo de compreensão. Estas ferramentas auxiliam o projetista na construção de modelos de projeto, através de mecanismos de análise, exploração e visualização da informação em diferentes níveis de abstração. As características destes mecanismos, genericamente referidos como técnicas de suporte, determinam a eficácia das ferramentas para adequar-se a diferentes atividades e domínios da aplicação [TIL 96].

As atividades inerentes do projeto de *software* se enquadram como atividades que se valem da experiência pessoal e da inteligência humana, portanto, a utilização de heurísticas deve prover auxílio para a solução de problemas característicos de qualquer projeto orientado a objeto, independente do domínio da aplicação.

As heurísticas estudadas determinam procedimentos de como modelar as classes, objetos, atributos, mensagens e os tipos de relacionamentos existentes.

2 Motivação

Segundo Casselman [CAS 92] e Harmon [HAR 93], as metodologias de desenvolvimento de aplicações orientadas a objetos, como por exemplo, as propostas por: Coad e Yourdon, Rumbaugh, Booch, Martin e Odell, Jacobson e Coleman, mostram técnicas variadas de modelagem e passos diferenciados para a construção de especificações de um sistema. Percebe-se, no entanto, a necessidade de definir processos com notações mais formais que conduzam a níveis elevados de garantia de qualidade dos modelos gerados. Surge, então, a necessidade de refinar e depurar ainda mais os processos de modelagem com a criação de regras, e padrões formais e não intuitivos.

É necessário considerar que o desenvolvimento de *software* orientado a objetos de boa qualidade não é uma tarefa que possa ser realizada facilmente por projetistas inexperientes em relação a tecnologia. Particularmente, o projeto de sistemas orientados a objetos é definitivamente mais complexo de ser realizado do que o projeto de sistemas que se utilizam da tecnologia procedimental convencional. Uma tendência comum entre os projetistas, é por exemplo, a sobreutilização de herança que conduz a programas inflexíveis e com difícil adaptação, ocasionando uma grande proliferação de classes.

A produção e manutenção de *software* ainda consomem muito esforço e apresentam um custo final elevado. Os pesquisadores Agarwal, Jiazhong, Schaschinger e Tse definem ferramentas para o auxílio a modelagem de sistemas com o objetivo de: melhorar a qualidade de projetos e diminuir os custos de produção.

O modelo definido em [AGA 95] propõe uma ajuda para o entendimento do processo de desenvolvimento de *software* orientado a objetos e avalia a manutenção de esforços despendidos no sistema e gerencia um processo de otimização. O trabalho apresenta técnicas e regras para a definição das estruturas (entidades e relacionamentos) dentro das etapas de desenvolvimento de *software*.

A proposta descrita em [JIA 96] caracteriza-se pela captura e simulação dinâmica de relacionamentos entre objetos. O trabalho cita uma série de passos para a modelagem de sistemas. Estes passos não são seqüenciais, e existe uma grande iteração e relacionamento entre os mesmos. Em [JIA 96] é citada uma representação formal para caracterizar o domínio do conhecimento, a qual torna mais práticos e precisos os processos de transformação durante a fase de modelagem.

Ferramentas com a capacidade de diagnosticar e interpretar padrões de projeto modificando situações quando necessário representam um avanço substancial no estado da arte em ferramentas de apoio ao projeto de *software* orientado a objetos. Neste sentido, a abordagem orientada a regras (heurísticas) adotada para a representação de padrões, pode ser utilizada como uma alternativa para suportar atividades desta natureza.

3 Heurísticas de Projeto

O desenvolvimento de *software* é visto como uma transformação de uma especificação informal, expressa em linguagem natural, para outra especificação, formal e sem ambigüidades, que possa ser processada pelo computador.

O projetista, através do uso de técnicas e metodologias, procura relacionar as características de uma linguagem natural (informal e imprecisa) com as exigidas pelos computadores (formal e consistentes). Neste sentido, pesquisadores procuram obter recursos que tornem a atividade de programação de computadores mais próxima das formas de raciocínio e expressão humanas. Sob este enfoque, os sistemas especialistas passam a ser estudados com maior ênfase e um dos maiores obstáculos que irão enfrentar, junto a construção de sistemas, é a amplitude do domínio deste tipo de aplicação.

A escolha por regras heurísticas para auxiliar a formalização de projetos, fundamenta-se no fato de que elas são adequadas para guiar o desenvolvimento de atividades as quais se valem da experiência pessoal, da observação e da inteligência humana. Desse modo, como as atividades de projeto se enquadram nessas características, a utilização de algoritmos para conduzir a realização desse tipo de tarefa não é plausível. As heurísticas definidas em [FRE 98] caracterizam-se por ajudar a solucionar problemas de qualquer projeto orientado a objeto, independente do domínio da aplicação.

4 Relacionamento entre Heurísticas e *Patterns*

Criou-se um relacionamento entre as heurísticas e *patterns* de projeto. Os *patterns* consistem de um padrão de projeto fonte (padrão ruim), uma heurística e um mapeamento para um projeto objetivo (padrão bom), aplicáveis a qualquer domínio de aplicação.

As heurísticas propostas, neste artigo, foram extraídas de decisões de projeto que se apresentam em diferentes domínios. Se estas decisões conduzem a projetos que possuem um certo grau de qualidade, como fácil entendimento, fácil manutenção e menor complexidade, é importante generalizá-las em regras. Freqüentemente, devido à inexperiência dos projetistas, são gerados projetos de nível de qualidade indesejável, o que faz interessante generalizar uma transformação padrão, ou seja, um *pattern* que capture um projeto ruim e transforme-o num bom projeto.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

5 Heurísticas para Manipulação de Atributos e Comportamento

Nesta seção são apresentadas heurísticas que visam apoiar a determinação de atributos, métodos e mensagens. As heurísticas descritas em [FRE 98] representam também contribuições para a modelagem de: relacionamentos de uso, relacionamentos contidos, heranças e associações.

As heurísticas são também enunciadas através de lógica no intuito de apresentá-las de maneira formal com vistas a sua utilização em uma ferramenta de especificação de projetos. Conta-se com o apoio de um dicionário de dados o qual armazena informações a respeito das classes envolvidas na aplicação. O dicionário foi construído na forma de tabelas e para acesso a estas dispõe-se da utilização de cálculo relacional de tuplas [ELM 89] e [KOR 94].

O cálculo relacional é uma linguagem não procedural a qual permite que se defina um conjunto de tuplas a partir de expressões do tipo { t | P(t) } onde para cada tupla t o predicado P(t) é verdadeiro: t ε Result → P(t). Uma variável tupla (VT) representa a cada instante uma tupla T de uma determinada relação R. Uma fórmula P(t) pode apresentar mais de uma variável.

Em uma determinada fórmula uma VT pode aparecer como: variável destino quando estiver associada a um quantificador existencial (∃) ou universal (∀) ou variável livre em caso contrário ao anterior.

Qualquer fórmula P(t) do cálculo relacional de tuplas é constituída de átomos. Um átomo pode assumir uma das seguintes formas: s ε R; s[x] comp u[y] onde comp ε {<, ≤, =, ≠, ≥, > } e s[x] comp C, onde C é constante no domínio de x.

Para formarem fórmulas, átomos podem ser combinados da seguinte maneira:

- um átomo é uma fórmula;
- se P1 é fórmula então ¬P1 e (P1) são fórmulas;
- se P1 e P2 são fórmulas então também o são: P1 v P2, P1 ^ P2 e P1 = P2;
- se P(s) é uma fórmula válida contendo uma VT livre s, então também o são: ∃s ε R(P(s)) e ∀s ε R(P(s)).

Devido a limitações do cálculo relacional que provê somente instruções de recuperação de informações em tabelas resolveu-se neste trabalho estender o cálculo proporcionando assim uma série de outros símbolos para utilização em procedimentos como por exemplo: entradas de dados (Entra Nome_Método, **Seleciona Caracterísitca_Método**), comparações (**Se Código_A ∩ Código_B ≠ Nill**), modificação de informações (Código_A - Código_Semelhante_Método ∪ Nome_Novo_Método) entre outros.

O dicionário de dados é apresentado na seguinte forma:

Nome_da_tabela (Nome_atributo1, Nome_atributo2, ... , Nome_atributok)

Dicionário
Classes (Nome_Classe, ...)
Atributos (Nome_Classe, Nome_Atributo, Tipo_Atributo , ...)
Métodos (Nome_Classe, Nome_Método,Tipo_Método, Código, Característica, ...)
MétodosxAtributos (Nome_Classe, Nome_Método, Nome_Atributo, ...)
MétodosxMétodos(Nome_Classe, Nome_Método_Chama, Nome_Método_Chamado, ...)

H1: Esconder os atributos públicos dentro da classe.

Esconder informações, nos níveis de projeto e implementação, traz um grande número de benefícios para a segurança do sistema [RUM 91] e [RUM 94]. Quando um atributo é público, há uma dificuldade para determinar que segmentos do sistema são dependentes do atributo.

Considere-se o exemplo em que o projetista necessita fazer a inserção de um novo atributo na classe. Conforme a Figura 1, o projetista admite o uso de *bloco* (novo atributo) como sendo público para as funções de acesso randômico (I/O), mas o que realmente faz-se necessário é a criação de operações para executar esta tarefa. Assim as operações *fseek* e *ftell* (padrões da linguagem C) não serão modificadas e, portanto, podem ser públicas.

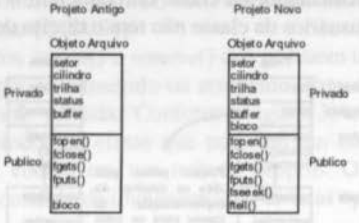

FIGURA 1 - Atributo público acidental.

Por outro lado, o atributo *bloco* será modificado pelas operações e, portanto, torna-se interessante ficar restrito somente ao objeto. Isto diminui o perigo de acontecerem erros, pois as informações podem ser manipuladas somente pelo próprio objeto.

Utilizando cálculo relacional estendido para a definição da heurística, tem-se:

∀ Classes ∈ Dicionário
{ c | c ∈ Classes }
{ t | ∃a ∈ Atributos (a [Nome_Classe] = c [Nome_Classe] ^
a [Tipo_Atributo] = Atributo_Público ^ t = a) }
t [Tipo_Atributo] = Atributo_Privado
Entra Nome_Novo_Método
{ s | ¬ ∃m ∈ Métodos (m [Nome_Classe] = c [Nome_Classe] ^
m [Nome_Método] = Nome_Novo_Método ^
s = m) }
Label
Seleciona Caracterísitca_Método (Get, Set)
Entra Código_Novo_Método
s [Nome_Método] = Nome_Novo_Método
s [Tipo_Método] = Método_Público
s [Código] = Código_Novo_Método
s [Carcterística] = Característica_Método
retorna label

O cálculo relacional estendido acima propõe a identificação de atributos públicos nas classes do sistema e prevê a modificação destes para privados proporcionando a criação de métodos *get* e *set* públicos para acesso aos referidos atributos.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

H2: Não devem ser definidos atributos protegidos na classe base.

É importante manter os atributos escondidos e acessá-los através de operações da classe [BOO 94]. No exemplo da classe fruta, Figura 2, é necessário verificar se *peso*, por exemplo, precisa ser definido como protegido. No caso da classe *Maçã*, é necessário que esta possua um método *imprime* próprio. Este método *imprime* precisa utilizar o atributo *peso*, criando a necessidade na classe *Maçã* de manusear este comportamento. No entanto, a classe *Fruta* não deve saber informação alguma a respeito da classe derivada *Maçã*.

A melhor solução para o caso é criar uma função protegida, chamada *get_peso()*, que simplesmente retorna o conteúdo do atributo *peso*. Neste caso os métodos de *Maçã* são dependentes apenas da interface protegida de *Fruta*, o que torna a implementação de mais fácil uso. Portanto, o implementador da classe derivada tem o direito de acessar os dados da classe base *Fruta*, mas os usuários da classe não tem o direito de usá-los.

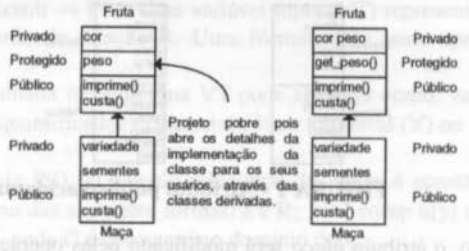

FIGURA 2 - Eliminando atributos protegidos

A utilização do atributo peso protegido pode vir a tornar mais fácil acomodar novas implementações, mas requer que os usuários dos dados protegidos examinem os métodos da classe derivada para possíveis modificações.

Utilizando cálculo relacional estendido para a definição da heurística, tem-se:

```
∀ Classes ∈ Dicionário
    { c | c ∈ Classes }
        { t | ∃a ∈ Atributos (a [ Nome_Classe ] = c [ Nome_Classe] ^
                              a [ Tipo_Atributo ] = Atributo_Protegido ^ t = a) }
        t [ Tipo_Atributo ] = Atributo_Privado
        Entra Nome_Novo_Método
        { s | ¬∃m ∈ Métodos (m [ Nome_Classe ] = c [ Nome_Classe]
                            m [ Nome_Método ] = Nome_Novo_Método ^
                            s = m) }
            Label
                Seleciona Caracterísitca_Método (Get ou Set)
                Entra Código_Novo_Método
                s [ Nome_Método ] = Nome_Novo_Método
                s [ Tipo_Método ] = Método_Protegido
                s [ Código ] = Código_Novo_Método
                s [ Carcterística ] = Característica_Método
                retorna Label
```

O cálculo relacional estendido acima propõe a identificação de atributos protegidos nas classes do sistema e prevê a modificação destes para privados proporcionando a criação de métodos *get* e *set* protegidos para acesso aos referidos atributos.

Algumas linguagens não suportam a notação de mecanismo de acesso a classes protegidas. Estas linguagens compensam esta falta fazendo o acesso privado comportar-se

como protegido ou forçando o usuário a colocar todos os membros protegidos na interface pública. A primeira solução desdobra as implementações das classes bases em classes derivadas, resultando num número muito grande de manutenções, e a segunda força o usuário a manipular detalhes de implementação através da interface pública. Em ambos os casos, não é possível capturar a essência da seção protegida no projeto.

H3: Não implementar detalhes na interface pública de uma classe.

Esta heurística tem como objetivo reduzir a complexidade da interface da classe para os usuários. A idéia básica é manter o usuário apto a utilizar os detalhes de algo, mas que não se torne necessário conhecer como estes detalhes foram implementados. É importante não criar, desordenadamente, na interface pública, itens que o usuário da classe não seja capaz de usar ou não se interesse em usar [COA 91].

Considera dois métodos *insere()* e *remove()* que possuem um código comum referente à localização e conexão do que será inserido ou removido. Este código comum foi colocado no método *procura()* na parte privada. Conforme Figura 3, cria-se um método comum privado, a partir de dois métodos da classe que possuam um código em comum, porque é conveniente encapsular este código em um método próprio. O novo método não é uma operação adicional, mas sim detalhes de implementação de duas operações de uma classe.

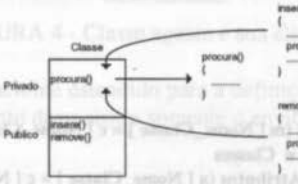

FIGURA 3 - Método privado com código comum.

Utilizando cálculo relacional estendido, demonstra-se a criação do método:

∀ Classes ∈ Dicionário
{ c | c ∈ Classes }
∀ Métodos ∈ Classes
{ m | ∃m ∈ Métodos (t | ∃n ∈ Métodos (m [Nome_Classe] = c [Nome_Classe]
n [Nome_Classe] = c [Nome_Classe]
n [Nome_Método] ≠ m [Nome_Método] ^
t = n)) }
Compara_Métodos(m [Código], t [Código])

Compara_Métodos (Código_A , Código_B)
Se Código_A ∩ Código_B ≠ Nill
Entra Nome_Novo_Método
{ u | ¬ ∃m ∈ Métodos (m [Nome_Classe] = c [Nome_Classe]
m [Nome_Método] = Nome_Novo_Método ^ u = m) }
u[Nome_Método] = Nome_Novo_Método
u [Tipo_Método] = Método_Privado
Código_Semelhante_Método = Código_A ∩ Código_B
u [Código] = Código_Semelhante_Método
u [Característica] = Código_Comum
Código_A - Código_Semelhante_Método ∪ Nome_Novo_Método
Código_B - Código_Semelhante_Método ∪ Nome_Novo_Método

O cálculo relacional estendido acima propõe para todas as classes a comparação de um método com todos os outros para prever possíveis porções de códigos comuns que

proporcionam a criação de um novo método privado. Para cada porção comum criada elimina-se dos métodos que deram origem esta porção e estes recebem uma chamada ao novo método.

H4: Manter ligados atributos e comportamento.

Faz-se importante que o usuário envie uma mensagem e a classe devolva as informações necessárias sem a necessidade de utilizar ou perguntar por determinados pré-requisitos.

A violação desta heurística poderá causar ao projetista transtornos na programação, ou seja, para realizar algum acesso ao sistema é necessário afetar o estado do sistema em dois ou mais lugares. Os dois lugares fazem parte da mesma identificação (chave de abstração) e, portanto, podem ser capturados na mesma classe. Muitas vezes, isto faz com que o objeto tenha que retirar informações de outros objetos através de uma função do tipo *get*.

Considerando, como exemplo, uma classe fogão que tenta aquecer o forno para cozer algum alimento, o usuário pode apenas perguntar se o forno encontra-se aquecido para que o sistema responda satisfatoriamente ou negativamente. Se for necessário, o usuário testar a temperatura, pressão ou outro elemento intrínseco do forno, estará incorrendo na violação da heurística.

Utilizando cálculo relacional estendido define-se quais atributos são utilizados pelos métodos da classe:

```
∀ Classes ∈ Dicionário
    { c | c ∈ Classes }
        ∀ Métodos ∈ Classes
          { s | ∃m ∈ Metodos (m [ Nome_Classe ] = c [ Nome_Classe] ^ s = m) }
              ∀ Atributos ∈ Classes
                { t | ∃a ∈ Atributos (a [ Nome_Classe ] = c [ Nome_Classe]^ t = a) }
                Verifica_Métodos_Atributos ( m, a )
        Verifica_Estatística ( MétodosxAtributos )

        Verifca_Métodos_Atributos ( m , a )
          Se a [ Nome_Atributo ] ε m [ Código ]
          { ma | ma ∈ MetodosxAtributos }
          ma [ Nome_Método ] = m [ Nome_Método ]
          ma [ Nome_Atributo ] = a [ Nome_Atributo ]

        Verifica_Estatística ( MétodosxAtributos )        ...
```

O cálculo relacional estendido acima propõe a identificação de atributos da classe pelos métodos que os utilizam, ou seja, são identificados todos os atributos utilizados em um determinado método para a definição de um controle estatístico final que definirá a ligação entre dados e comportamento.

H5: Eliminar classes que estão fora do sistema.

Uma classe que está fora do sistema é uma classe irrelevante ao domínio do sistema, mas nem sempre é fácil detectá-la. Durante sucessivas interações de projeto, torna-se eventualmente claro que algumas classes não requerem métodos, portanto, são classes que estão fora do sistema. Geralmente, estas classes enviam mensagens para o sistema mas não recebem mensagens de outras classes do domínio.

Durante a fase de análise é conveniente modelar o mundo real tanto quanto o possível. Na fase de projeto, modificações no modelo devem ser realizadas no intuito de adequar o

sistema aos recursos de produção de *software* disponíveis [COL 94]. As classes agentes são classes que disponibilizam tarefas na fase de análise e tendem a desaparecer na fase de projeto.

Considerando a Figura 4, que representa o envio de mensagens entre *Livro*, *Bibliotecário* e *Prateleira*. Qual o uso da classe *Bibliotecário* no sistema? *Bibliotecário* é uma classe irrelevante, cuja função é aceitar mensagens de *Livro* e *Prateleira* e reenviá-las ao destino desejado, ou seja, os métodos de *Bibliotecário* são mensagens para as outras classes. Em muitos casos, é possível eliminar as classes irrelevantes, no caso *Bibliotecário*. Esta classe captura a utilidade da abstração e isto deve ser levado em conta para a criação de *Livro* e *Prateleira*.

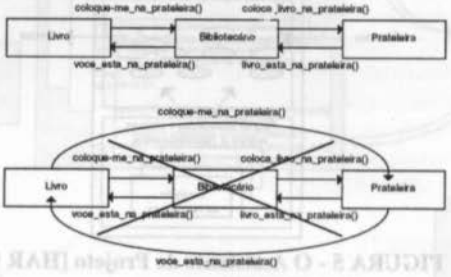

FIGURA 4 - Classe agente e sua eliminação

Utilizando cálculo relacional estendido para a definição da heurística, procura-se todos os métodos cujo comportamento desmonstre somente o envio de mensagens:

∀ Classes ∈ Dicionário
{ c | c ∈ Classes }
∀ Métodos ∈ Classes
{ m | m ∈ Métodos (t | ∃n ∈ Métodos (n [Nome_Classe] = c [Nome_Classe]
n [Nome_Método] ≠ m [Nome_Método] ^
t = n }
Verifica_Métodos_Métodos(m, t)
Verifica_Estatística (MétodosxMétodos)

Verifca_Métodos_Métodos (m , t)
Se t [Nome_Método] ε m [Código]
Se m [Código] – t [Código] = ∅
{ mm | mm ∈ MetodosxMetodos }
mm [Nome_Método_Chama] = m [Nome_Método]
mm [Nome_Método_Chamado] = t [Nome_Método]

Verifica_Estatística (MétodosxMétodos) ...

O cálculo relacional estendido acima propõe a identificação de métodos da classe que somente enviem chamadas a outros métodos, ou seja, são identificados todos os métodos que não disponibilizam um comportamento significativo.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDF

6 Assistente de Projeto

Um assistente de projeto é uma ferramenta de auxílio ao projetista para diagnosticar e verificar possíveis modificações de projeto, com o intuito de melhorar a qualidade do mesmo. A Figura 5, ilustra a estrutura de uma ferramenta que dispõe de um módulo de manipulação dos elementos envolvidos como classes, objetos e relacionamentos. Este módulo pode ser ativado via interface gráfica ou textual e conta com opções de manipulação, persistência, remoção e edição, entre outras.

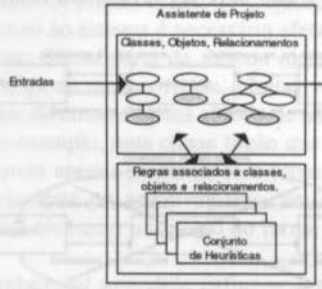

FIGURA 5 - O Assistente de Projeto [HAR 93]

O segundo módulo define as regras que são aplicadas aos elementos criados. As regras traduzem as heurísticas estudadas neste trabalho. As heurísticas estão implementadas em Prolog++ para Windows [VAS 95] e [WES 96] como classes que definem um determinado comportamento de acordo com as suas características, isto é, algumas advertem o projetista sobre algum problema, outras advertem e propõem modificações.

A ferramenta assistente de projeto, ilustrada na Figura 6 [FRE 98], caracteriza-se pela existência de cinco blocos principais: um módulo extrator de informações, um módulo responsável pela aplicação das regras, um módulo de geração de aplicações gráficas, um módulo de visualização e um módulo de geração de código.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

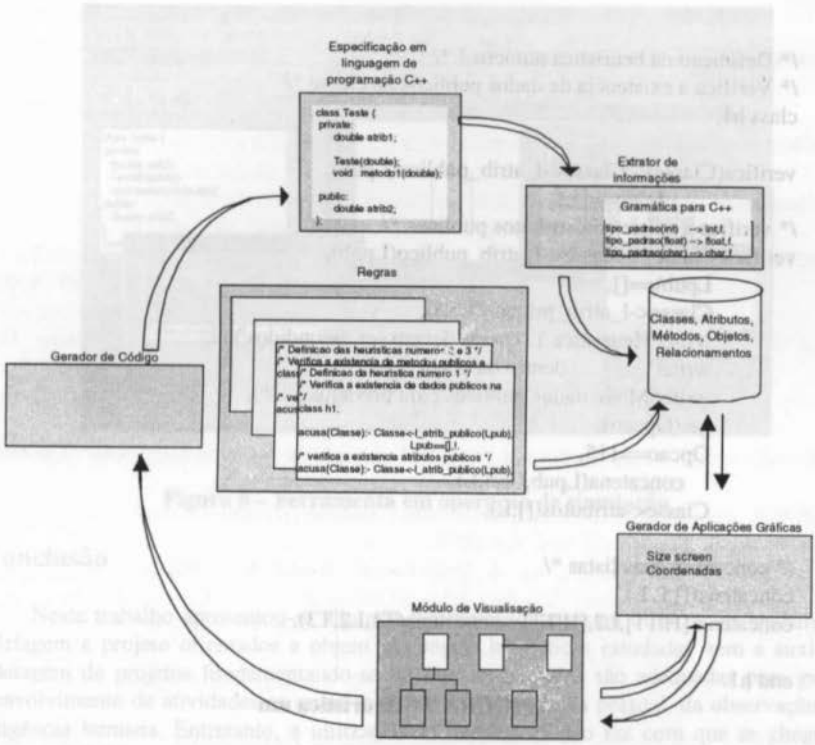

FIGURA 6- Ferramenta assistente de projeto

A partir da descrição fonte, produzida em alguma linguagem de programação, pelo próprio projetista ou por alguma ferramenta CASE, o módulo de extração de informações tem por objetivo extrair do texto as informações relevantes para a aplicação das heurísticas existentes. O módulo de extração cria uma base de dados (um Dicionário) com estas informações.

O módulo responsável pela aplicação das regras aplica sobre a base de dados as heurísticas existentes proporcionando, por vezes, modificações nas informações existentes. Estas informações são devidamente alteradas de acordo com o conhecimento do projetista.

A definição da classe *h1*, ilustrada na figura 7, representa a heurística H1, a qual recomenda que os atributos devem ser escondidos dentro da classe. A classe *h1* adverte o projetista quando existem atributos públicos e, ainda, propõe a transformação destes para privados. A transformação se dá pela concatenação de Lpub (lista de atributos públicos) a Lpri (lista de atributos privados) através do método concatena. A classe *h1*, quando chamada através da mensagem *verifica*, recebe como parâmetro a referência a uma classe (identificada pela variável lógica *Classe*), a qual deseja-se submeter a teste. Com esta referência, *h1* identifica os atributos públicos da referida classe através de pesquisa na lista Lpub, se esta é vazia ou não. Se existirem elementos na lista, a ferramenta questiona o projetista da possível concatenação aos atributos privados (lista Lpri).

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDF

```
/* Definicao da heuristica numero 1 */
/* Verifica a existencia de dados publicos na classe */
class h1.

verifica(Classe):- Classe<-l_atrib_publico(Lpub),
                   Lpub==[],!.
/* verifica a existencia atributos publicos */
verifica(Classe):- Classe<-l_atrib_publico(Lpub),
        Lpub\==[],
        Classe<-l_atrib_privado(Lpri),
        write('Heuristica 1: Dados devem ser escondidos'),nl,
        write('          dentro da classe.'),nl,
        write('Move dados publicos para privados .... ?'),
        get(Opcao),
        Opcao==115,
          concatena(Lpub,Lpri,L),
        Classe<-atributos([],L).

/* concatena duas listas */
concatena([],L,L).
concatena([H|T1],L2,[H|T3]):-concatena(T1,L2,T3).

end h1.
```

FIGURA 7 - Heurística um

Os módulos de geração de aplicações gráficas e de visualização permitem ao projetista verificar de forma gráfica a especificação textual, bem como manipular esta especificação, proporcionando ajustes caso necessário.

O módulo de geração de código tem por função, também, proporcionar alterações no código fonte em função de modificações proporcionadas pelas heurísticas ou pelo projetista.

A Figura 8 mostra o protótipo da ferramenta durante operação de simulação sobre as informações disponíveis. A ferramenta encontra-se em fase preliminar de construção, dispondo de algumas heurísticas e contando como interfaces os módulos de extração, edição e de aplicação das heurísticas definidas.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

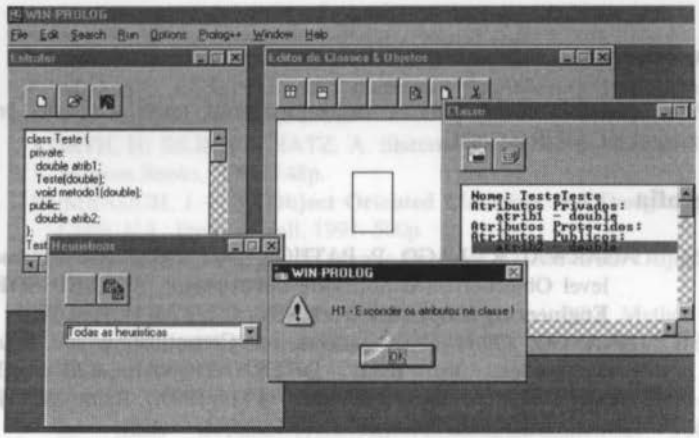

Figura 8 – Ferramenta em operação de simulação

7 Conclusão

Neste trabalho apresentou-se algumas heurísticas que visam otimizar os trabalhos de modelagem e projeto orientados a objeto. As regras heurísticas estudadas vem a auxiliar a modelagem de projetos fundamentando-se no fato de que elas são adequadas para guiar o desenvolvimento de atividades as quais se valem da experiência pessoal, da observação e da inteligência humana. Entretanto, a utilização de heurísticas não faz com que se chegue a respostas definitivas para os problemas em questão, mas estabelece algumas idéias para otimizar a construção de projetos.

Admite-se a possibilidade de implementação para parte das heurísticas estudadas. Em alguns casos acredita-se que limitar quantitativamente determinadas informações não representa uma boa idéia pois cada aplicação possui características e necessidades próprias a serem desenvolvidas [FRE 98].

O estudo desenvolvido é uma etapa que representa um esforço de pesquisa em direção à criação de diretivas para a otimização de projeto. Outro aspecto que precisa ser explorado para que se atinja esta meta é pesquisar um maior número de *patterns* de transformação com o objetivo de criar protótipos genéricos que possam enquadrar-se a projetos orientados a objeto.

Outra questão importante é o aprimoramento da ferramenta que permite guiar o usuário a passar de um projeto ruim para um projeto de boa qualidade. Tal ferramenta requer um estudo de propriedades tais como flexibilidade, portabilidade, eficiência, relacionamento *hardware/software* e importância da minimização de classes. Estas propriedades devem ser escalonadas e determinadas em grau de importância.

Finalmente, crê-se que o trabalho desenvolvido poderá ser útil para os estudos de construção de ferramentas que auxiliem o projetista em suas atividades e para trabalhos envolvidos na construção de protótipos de sistemas capazes de transformar automaticamente especificações de projeto.

Agradecimentos

Pelo apoio financeiro para a realização deste trabalho faz-se presesente os agradecimentos a CAPES E CNPQ.

Bibliografia

[AGA 95] AGARWAL R.; LAGO, P. PATHOS - A Paradigmatic Approach To High-level Object-oriented Sofftware development. **ACM SIGSOFT Software Engineering Notes**, New York, v.20, n. 2, p.36-41, Apr. 1995.

[ARA 93] ARANGO, G. et al. A process for Consolidating and Reusing Design Knowledge. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 13., 1993, Baltimore. **Proceeding...** California: IEEE Press, 1993.

[BOO 94] BOOCH, G. **Object-Oriented Ananlysis and Design with Aplications**. Redwood City: Benjamin Cummings, 1994. 589p.

[BAX 92] BAXTER, I. Design Maintenance Systems. **Comunications of the ACM**, New York, v.35, n.4. Apr.1992.

[CAS 92] CASSELMAN, R. **A Role-Based Model for Object-Oriented Design**. Vancouver, Canadá: Carleton University, 1992. Master Thesis.

[COA 91] COAD, P.; YOURDON, E. **Object-Oriented Analysis**. Englewood Clifs: Prentice-Hall, N.J, 1991. 233p.

[COA 92] COAD, P.; YOURDON, E. **Análise Baseada em Objetos**. Rio de Janeiro: Campus, 1992. 225p.

[COL 94] COLEMAN, D. et al. **Objetct-Oriented Development:** The Fusion Method. Englewood Clifs: Prentice-Hall, 1994.

[DEC 92] DE CHAMPEAUX, D.; LEA, D.; FAURE, P. The Process of Object Oriented Design. **ACM SIGPLAN Notices**, New York, v.27, n. 10, p.45-62, October 1992. Trabalho apresentado na Annual Conference on Object-Oriented Programming, Systems, Languages and Aplications - OOPSLA'92, 7, 1992, Vancouver.

[DEC 93] DE CHAMPEAUX, D.; LEA, D.; FAURE, P. **Object-Oriented System Development**. Reading: Addison-Wesley, 1993. 532 p.

[ELM 89] ELMASRI, N; NAVATHE, S.B. **Fundamentals of Database Systems.** Redwood City: Benjamin Cummings, 1989. 802p.

[FRE 96] FREITAS, A. **Um Estudo das Metodologias de Análise e Projeto Orientados a Objeto:** Trabalho Individual. Porto Alegre: CPGCC da UFRGS, 1996. (TI-572).

[FRE 98] FREITAS, A. **Heurísticas para Apoioar a Modelagem de Sistemas Orientados a Objeto.** Porto Alegre: CPGCC da UFRGS, 1998. 103p. Dissertação de Mestrado.

[HAR 93] HARMON, P. **Intelligent Software Systems Development**. New York: John Willey, 1993. 472p.

[JAC 94] JACOBSON, I. et al. **Object-Oriented Software Enginnering.** Reading: Addison-Wesley, 1994. 528p.

[JAC 95] JACOBSON, I. et al. **The Object Advantage:** Bussines Process Re-Engineering whit Object Technology. Reading: Addison-Wesley, 1995. 600p.

with optimization using a watermarked evaluation copy of CVISION PDFCompressor

[JIA 96] JIAZHONG, Z.; ZHIJIAN W. NDHORM: An OO Approach to Requirements Modeling. **ACM SIGSOFT Software Engineering Notes**, New York, v.21, n. 5, p.65-69, Sept. 1996.

[KOR 94] KORTH, H; SILBERSCHATZ, A. **Sistema de Banco de Dados.** São Paulo, Makron Books, 1994. 748p.

[RUM 91] RUMBAUGH, J. et al. **Object Oriented Modeling and Design**. Englewood Cliffs, N.J.: Prentice Hall, 1991. 500p.

[RUM 94] RUMBAUGH, J. et al. **Modelagem e Projetos Baseados em Objetos**. Rio de Janeiro: Campus, 1994. 655p.

[SCH 92] SCHASCHINGER H. ESA - An Expert Supported OOA Method an Tool. **ACM SIGSOFT Software Engineering Notes**, New York, v.17, n. 2, p.50-56, Apr. 1992.

[TIL 96] TILEY, S; SANTANA P. Towards a Framework for Program Understanding. In: IEEE INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, 4., 1996, Berlin, Alemanha. **Proceedings...** Los Alamitos: IEEE Press, 1996.

[TSE 94] TSE T. et al. The application of Prolog to Structured Design. **Software - Practise and Exeperience**, New York, v.24, n. 7, p. 659-676, July 1994.

[VAS 95] VASEY, P. **LPA-Prolog++:** Programmer's Reference. London: Logic Programming Associates Ltd, 1995. 162p.

[WES 96] WESTWOOD, D. **LPA-Prolog:** Thecnical Reference. London: Logic Programming Associates Ltd, 1996. 338p.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor