Re-Engenharia de Software, um Estudo de Caso

Julio Cesar Sampaio do Prado Leite[§]
Ana Paula Moreira Franco[¶]
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente, 225.
Rio de Janeiro 22453

Julho/91

Resumo

A re-engenharia de software começa a despontar como área de interesse principalmente porque possibilita um novo enfoque para o velho problema da manutenção. A re-engenharia é um processo distinto da engenharia tradicional, porque parte de um desenho já existente. Tendo como ponto de partida um desenho, a re-engenharia pode explorar oportunidades de reutilização. Como normalmente não se dispõe de um desenho que reflita o artefato que se quer refazer, a recuperação de desenho ou engenharia reversa é fundamental. A re-engenharia pode trazer ganhos de produtividade, principalmente, quando é necessário aumentar a funcionalidade de um artefato de software. Nosso artigo propõe uma estratégia de re-engenharia baseada na metodologia JSD (Jackson Systems Design). Através de um estudo de caso, aumento de funcionalidade de um sistema de hipertexto, exemplificamos a estratégia.

1 Introdução

Um fato que é consenso na comunidade de engenharia de software é a de que a manutenção de software resulta em comprometimento de grande parte dos recursos alocados ao ciclo de vida de um software. A literatura de manutenção de software, conforme relata Souza [Souza 91], é unânime nesta constatação. Em função desta realidade, propostas que possam reduzir estes custos têm grande importância.

O problema da manutenção agrava-se com o passar do tempo, na medida em que mais sistemas entram em manutenção e na medida em que cada vez mais cresce a demanda por recursos computacionais de suporte a sistemas de informação. Em função da necessidade de prover métodos eficazes para o desenvolvimento de software, e da dificuldade desse problema, o aspecto específico de manutenção é, muitas vezes, relegado a um segundo plano [Parikh 88, Vallabhanen 87]. O interesse crescente por métodos de

[§]Apoio da SCT-PR e do Cnpq.
[¶]Apoio da Faperj.

apoio à manutenção, fez com que a comunidade de pesquisa procurasse investigar este aspecto. A maior parte destas pesquisas utiliza-se de alguma forma de engenharia reversa e reutilização para aumentar a produtividade da tarefa de manutenção [Biggerstaff 89] [Chikofsky 90] [Rugaber 90] [Baxter 90].

Neste artigo descreveremos uma estratégia de recuperação que utiliza o JSD [Jackson 83] como forma de registro do artefato alvo da re-engenharia. Esta estratégia é exemplificada por um caso real, em que utilizamos, com sucesso, essa estratégia. O artigo está organizado em cinco Seções. Na Seção 2 apresentamos nosso conceito de re-engenharia de software e sua ligação com a reutilização. Na Seção 3 descrevemos o HyperLex[1] e sua história. Na Seção 4 apresentamos a estratégia utilizada e sua instanciação para o HyperLex. Concluímos, Seção 5, ressaltando o que aprendemos na condução do estudo de caso, bem como delineando trabalhos futuros com o objetivo de prover assistência automatizada ao processo de re-engenharia de software.

2 Re-Engenharia de Software

A re-engenharia de um artefato de software é composta de duas partes distintas:

- recuperação do desenho e
- reutilização do desenho na tarefa de re-construção (re-desenho).

A recuperação do desenho, segundo Biggerstaff [Biggerstaff 89], recria abstrações de desenho segundo uma combinação de: código, documentação existente, experiência pessoal, e conhecimento geral do problema e da aplicação. A recuperação de desenho caracteriza a engenharia reversa, isto é, a arquitetura de um sistema é montada a partir de um processo de arqueologia. Este processo é claramente um processo *bottom-up* onde aspectos de modelagem referentes a agregação, generalização e associação estão envolvidos. Normalmente, a principal fonte de informação é o código, sendo que, a disponibilidade de outras fontes é dependente de cada caso em particular. Leite em [Leite 91] descreve uma situação particular; a recuperação do protótipo Draco, onde além da carência de informações complementares, não se dispunha do ambiente de programação para executar o código a ser recuperado.

A reutilização do desenho no processo de re-construção do artefato acontence de acordo com as necessidades de nova funcionalidade do artefato. Esta reutilização, não, necessariamente, implica na reutilização total da arquitetura recuperada, nem do código que a implementa. A reutilização a nível de desenho, permite uma maior liberdade na escolha das implementações, isto é: o código existente pode ser inteiramente reutilizado, ou modificado, ou substituído por novo código.

O processo de re-desenho além de utilizar-se das informações constantes do desenho recuperado utiliza informações proveniente do Universo de Informações[2]. Estas informações

[1] ©JCSPL, APMF

[2] Universo de informações é o contexto geral no qual o software deverá ser desenvolvido. O universo de informações inclui todas as fontes de informação e todas as pessoas relacionadas ao software. Estas pessoas são também conhecidas como os atores desse universo. O universo de informações é a realidade circunstanciada pelo conjunto de objetivos definidos pelos que demandam o software.

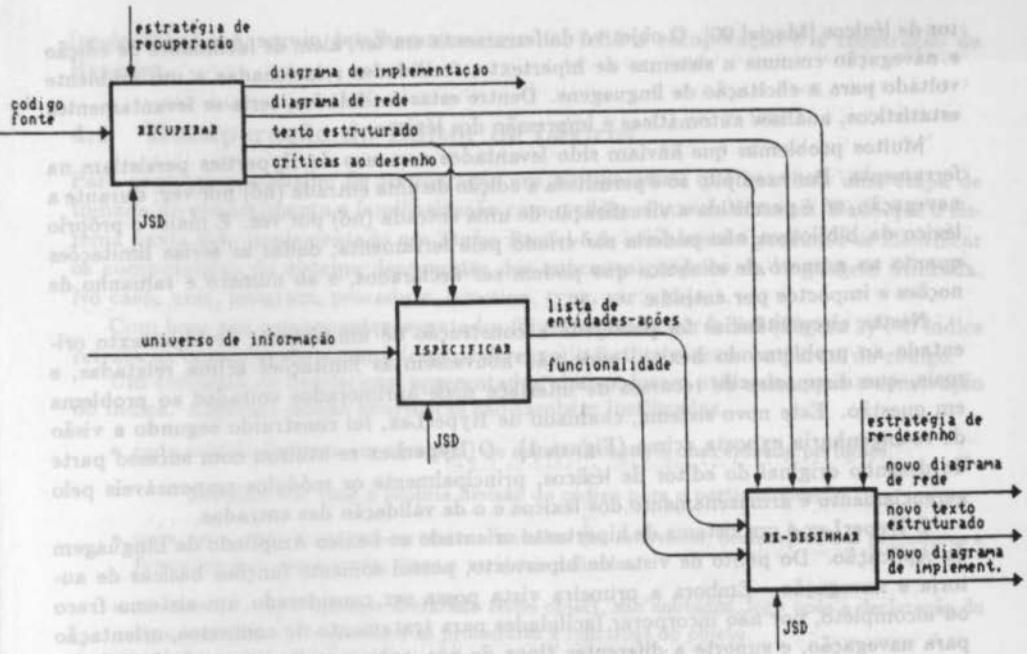

Figura 1: Re-Engenharia de Software

são modeladas e utilizadas como entrada no processo de re-desenho. Na Figura 1 utilizamos um actigrama SADT [Ross 77] para mostar uma instanciação de um processo de re-engenharia.

3 Hyperlex, o Caso Estudado

Uma das atividades do nosso grupo de pesquisa é a investigação de métodos e técnicas para suporte ao processo de análise de domínios. Uma das estratégias usadas é uma redução do problema da análise de domínios. Dentro dessa idéia se coloca o estudo de linguagens da aplicação [Leite 89]. Um dos métodos que vem sendo experimentado para a elicitação de linguagens da aplicação tem por base o Léxico Ampliado da Linguagem (LAL) [Leite 90]. O objetivo deste léxico é descrever precisamente a semântica dos símbolos da linguagem em questão usando a própria terminologia da aplicação. Assim, cada símbolo constitui uma entrada do léxico, a qual contém noções e impactos associados ao símbolo. Estas características gerais mostraram que um hiperdocumento era uma forma natural para representar o LAL: as entradas do léxico constituem os nós; os símbolos referenciados, dado o uso do *princípio da circularidade* [Leite 90], constituem os elos do hiperdocumento.

Com base nestas idéias do Léxico Ampliado da Linguagem foram realizadas duas experiências. Uma foi a criação de um léxico ampliado da linguagem sob a forma de um hiperdocumento utilizando o Hyperties [Leite 90]. A outra foi a implementação de um edi-

tor de léxicos [Maciel 90]. O objetivo da ferramenta era ter, além de facilidades de edição e navegação comuns a sistemas de hipertexto, facilidades relacionadas a um ambiente voltado para a elicitação de linguagens. Dentre estas facilidades, teria-se levantamentos estatísticos, análises automáticas e impressão dos léxicos.

Muitos problemas que haviam sido levantados com uso do Hyperties persistiam na ferramenta. Por exemplo: só é permitida a edição de uma entrada (nó) por vez; durante a navegação, só é permitida a visualização de uma entrada (nó) por vez. E mais: o próprio léxico da biblioteca não poderia ser criado pela ferramenta, dadas as sérias limitações quanto ao número de símbolos que podem ser declarados, e ao número e tamanho de noções e impactos por entrada.

Nestas circunstâncias foi planejada a construção de um sistema de hipertexto orientado ao problema do Léxico, onde não houvessem as limitações acima relatadas, e mais, que dispusesse de recursos de interface mais aprimorados voltados ao problema em questão. Este novo sistema, chamado de HyperLex, foi construído segundo a visão de re-engenharia exposta acima (Figura 1). O HyperLex re-utilizou com sucesso parte do desenho original do editor de léxicos, principalmente os módulos responsáveis pelo gerenciamento e armazenamento dos léxicos e o de validação das entradas.

O HyperLex é um sistema de hipertexto orientado ao Léxico Ampliado da Linguagem da Aplicação. Do ponto de vista de hipertexto, possui somente funções básicas de autoria e navegação. Embora a primeira vista possa ser considerado um sistema fraco ou incompleto, por não incorporar facilidades para tratamento de contextos, orientação para navegação, e suporte a diferentes tipos de nós, cabe ressaltar que tais facilidades são dispensáveis à aplicação em questão, ou seja, a construção de LAL's.

4 A Re-Engenharia no Caso do HyperLex

A re-engenharia utilizada no caso HyperLex teve como ponto de partida a experiência obtida com a recuperação do protótipo Draco [Leite 91]. No caso de Draco foi utilizada uma combinação de inspeções, JSD, e prototipação. No caso do editor de léxicos não foi utilizada a prototipação e a inspeção aconteceu com um acoplamento fraco. A parte substancial da experiência foi a utilização do JSD como forma de registrar a recuperação do desenho.

JSD [Jackson 83] é uma metodologia que vem despertando o interesse de pesquisadores nacionais [Ambrosio 89, Rodriguez 90, Prado 91]. Ela distingue duas importantes tarefas: o desenvolvimento da especificação, e o desenvolvimento da implementação. A tarefa de desenvolvimento da especificação é composta de duas etapas: a especificação do modelo da realidade e a especificação das funções do sistema. A metodologia JSD compreende 3 estágios: o estágio de modelagem, o estágio de rede e o estágio de implementação. Dentro desses estágios convivem 5 fases: entidade/ação, estrutura/entidade, modelo inicial, função, cronologia e implementação.

A metodologia JSD prescreve um desenvolvimento linear da fase entidade/ação à fase implementação. No caso da recuperação a ordem dessa sequência é invertida. Parte-se do código, identifica-se cada parte do software, monta-se o diagrama de implementação, recupera-se o texto estruturado dos processos, e monta-se o diagrama de rede (modelo inicial mais funções). Uma vez disponível, o desenho serve de base para um novo desenho

(re-desenho). A seguir detalharemos como foi feita a recuperação e a construção do Hyperlex.

4.1 Recuperação do Editor de Léxicos

Para começar o trabalho de recuperação era indispensável que houvesse uma etapa de tomada de conhecimento e familiarização com o código fonte do sistema. Dado que o sistema havia sido implementado em Turbo Pascal 5.5, inicialmente procurou-se identificar os componentes do sistema decorrentes dos conceitos próprios da linguagem utilizada. No caso, unit, program, procedure, function, type, var e object.

Com base nos componentes levantados foi estruturado um índice (Figura 2)[3]. O índice retrata as conexões do sistema (cross-reference) e facilita o acesso a partes do código.

Um conjunto de heurísticas, apresentadas abaixo, foram utilizadas para a construção do índice. Algumas dessas heurísticas são também justificadas.

- cada módulo (arquivo em separado) do sistema constitui uma entrada no índice;
  - Justificativa: usar a própria divisão do código para o particionamento.
- para cada módulo anotam-se: units e includes que referencia; declarações das procedures e functions que compreende; tipos e variáveis definidos.
- no caso de existir um tipo declarado como object, são anotados, logo após a declaração do tipo, variáveis de instância e as procedures e functions do objeto.

Finalizando esta fase de reconhecimento do código fonte, foi feito o cross-reference das procedures e functions. O objetivo desta atividade era o de recuperar a estrutura do sistema. Uma vez que havia o entendimento do código, iniciou-se o uso do JSD na recuperação do desenho. Dado que a estratégia definida era o emprego da metodologia de maneira inversa, a primeira fase a ser seguida era fase de implementação.

O JSD propõe o diagrama de implementação e o texto estruturado dos processos, funções e escalonador do sistema como produtos desta fase. Na recuperação, no entanto, consideramos o próprio código fonte do sistema como sendo o texto estruturado, e portanto, a atenção nesta etapa estava centrada na recuperação do diagrama de implementação.

Para a elaboração do diagrama de implementação (Figura 3) foi usado o seguinte conjunto de heurísticas e justificativas, quando for o caso:

- cada objeto declarado constitui um processo de implementação;
  - Justificativa: supõe-se que o encapsulamento de objeto foi correto, e utiliza-se a metáfora de entidade-JSD/objeto.
- cada unit declarada constitui um processo de implementação;
  - Justificativa: supõe-se que uma unit caracteriza um particionamento de alto nível de abstração.
- o objeto onde está declarado o main constitui o processo escalonador[4];

[3] No índice aparecem palavras em inglês e português, isto acontece porque além das palavras chaves da linguagem de implementação, o sistema recuperado mistura inglês e português como linguagem natural.

[4] No caso do editor de léxico, o objeto (Pascal 5.5) usuário contém a declaração da procedure main.

```
-ÍNDICE: Editor de Lexicos----------------------------------------------------

Arquivo - DEFINIR.PAS

    Units      -  (não tem)
    Includes   -  (não tem)
    Types         Symbol      : string
                  Symbols     : array of Symbol
                  Notion      : string
                  NotionList  : array of Notion
                  Impact      : string
                  ImpactList  : array of Impact
                  Entry       : record
                                  Cj Simb   : Symbols
                                  Nocoes    : NotionList
                                  Impactos  : ImpactList
                  Position    : ^EntryList
                  EntryList   : record
                                  Entrada   : Entry
                                  Next      : Position
                  SymbolTable: array of record
                                         Simbolo : Symbol
                                         Posicao : Position
                  SymbolTablePtr = ^SymbolTable
                  TablePosition  : integer

-----------------------------------------------------------------------------

Arquivo - DICIONAR.PAS

    Units      -  (não tem)
    Includes      DEFINIR.PAS
    Types         object dictionary
                     var Entrada_Atual: Entry
                         Posicao-Atual: Position
                         Dados:         Position
                         Tabela:        SymbolTablePtr
                         TableSize:     TablePosition
                         Arquivo:       string
                  proc Inicializa(FileName: string)
                  proc Termina
                  func FindSymbol(Simbolo: Symbol) : TablePosition
                  func Symbol_in_Entry(Simbolo: Symbol;
                                       Entrada: Entry) : boolean
                  proc Include_Symbol(Simbolo: Symbol; Posicao : Position)
                  proc Exclude_Symbol(Simbolo: Symbol)
                  proc Criar_Entrada
                  proc Destruir_Entrada(Simbolo: Symbol)
                  proc Alterar_Entrada(Posicao: Position; Entrada: Entry)
                  proc Procurar_Simbolo(Simbolo: Symbol; var Entrada: Entry;
                                        var Posicao: Position)
                  proc Reagatar_Posicao(Posicao:Position; var Entrada: Entry)
                  proc Relatorio_Simbolo
                  proc Relatorio_Entrada

    Vars          dicionario: dictionary

obs: as referências a dicionario serão tratadas por lexico
```

Figura 2: índice

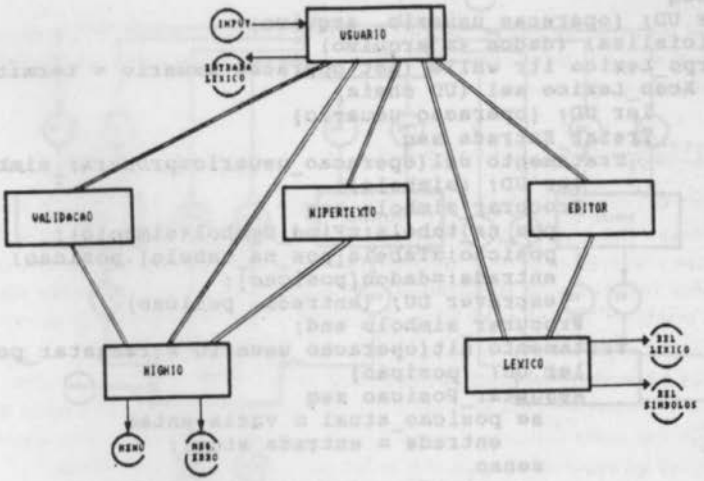

Figura 3: Diagrama de Implementação Recuperado

  - Justificativa: o main, geralmente, coordena a chamada dos outros procedimentos do sistema.

- a interface entre os processos é definida com base no cross-reference e na estrutura do código;
- cada arquivo gerado ou consultado pelo sistema constitui um dos arquivos do diagrama e está ligado aos processos responsáveis pelo seu tratamento;
- os relatórios e mensagens de erro do sistema constituem as saídas do diagrama;
- as entradas do sistema são agrupadas em uma única fila de mensagem que é lida pelo escalonador.

  - Justificativa: simplificar a elaboração do diagrama de implementação.

A etapa anterior à fase de implementação no JSD, a fase de rede, compreende a elaboração do diagrama de especificação e do texto estruturado dos processos. Assim, na recuperação desta etapa, foi feito um resumo de todo o programa, o que resultou no texto estruturado dos processos. O objetivo deste resumo era a recuperação dos passos executados, ou seja, do comportamento do sistema. A Figura 4 é um exemplo do texto estruturado do processo chamado Léxico.

Para a elaboração do texto estruturado foi usada a seguinte estratégia:

- cada processo do diagrama de implementação tem sua lógica detalhada através de um texto estruturado usando os componentes básicos previstos pela metodologia JSD: sequência, seleção, iteração. O primeiro processo a ser tratado é o escalonador;

```
Lexico
Lexico seq
     ler UD; (operacao_usuario, arquivo)
     Inicializa; (dados <= arquivo)
     Corpo_Lexico itr while (not operacao_usuario = termina)
        Acao_Lexico sel (UD cheia)
           ler UD; (operacao_usuario)
           Tratar_Entrada seq
             Tratamento sel(operacao_usuario=procurar_simbolo)
                ler UD; (simbolo)
                Procurar_simbolo seq
                  pos_na_tabela:=Find_Symbol(simbolo);
                  posicao:=Tabela[pos_na_tabela].posicao;
                  entrada:=dados[posicao];
                  escrever DU; (entrada, posicao)
                Procurar_simbolo end;
             Tratamento alt(operacao_usuario = resgatar_posicao)
                ler UD; (posicao)
                Resgatar_Posicao seq
                   se posicao_atual = vazia entao
                      entrada = entrada_atual;
                   senao
                      entrada = dados[posicao].entrada;
                   escrever DU; (entrada)
                Resgatar_Posicao end:
             Tratamento alt (operacao_usuario = criar_entrada)
                   ...
             Tratamento alt (operacao_usuario = destruit_entrada)
                   ...
             Tratamento end;
          Tratar_entrada end;
       Acao_Lexico alt (ED cheia)
          ler ED; (operacao_edicao)
          Editar_Entrada seq
            Edicao sel (operacao_edicao = ler_cj_simbolos)
               escrever DE; (entrada_atual.cj_simb)
            Edicao alt (operacao_edicao = gravar_novo_cj_simbolos)
               ler ED; (simbolos)
               entrada_atual.cj_simb := simbolos
            ...
            Edicao alt (operacao_edicao = alterar_entrada)
               ler ED; (posicao_editada, entrada_editada);
               Alterar_Entrada;
            Edicao end;
          Editar_Entrada end;
       Acao_Lexico end;
    Corpo_Lexico end;
    Termina; (arquivo <= dados)
 Lexico end;
```

Figura 4: Texto Estruturado Recuperado

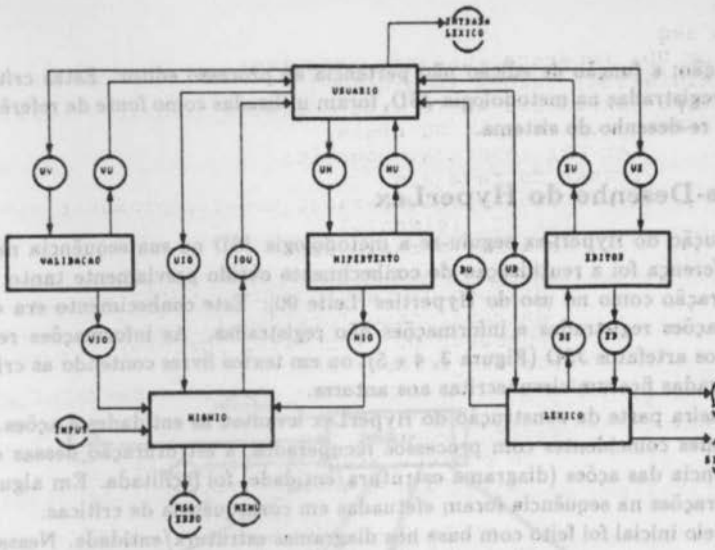

Figura 5: Diagrama de Rede Recuperado

- cada chamada a uma procedure ou function está representada no texto estruturado por um comando de escrita, numa fila de mensagem que liga o processo onde ocorreu a chamada ao processo chamado;
- o retorno de functions e os parâmetros passados por referência a procedures e functions estão representados por um comando de escrita no processo chamado e um comando de leitura no processo que chamou, na fila de mensagem que liga os dois processos;
- o acesso a um arquivo tratado por um outro processo está representada pelo comando de leitura do vetor de estado do processo responsável pelo tratamento do arquivo;
- as saídas do diagrama de implementação são mantidas;
- a entrada do diagrama de implementação, dependendo de como foi implementada a interface, será dividida em várias filas de mensagem, e ligada aos processos correspondentes do modelo de rede.

É importante notar que não ocorreu um tratamento mais detalhado das entradas e saídas do sistema, pois estas representam aspectos fundamentalmente ligados a interface, a qual não seria aproveitada no novo sistema. Deste modo, estes aspectos foram suprimidos do resumo.

De posse dos textos estruturados dos processos, foi fácil a elaboração do diagrama de rede. Os processos são os mesmos do diagrama de implementação e as filas de mensagem e vetores de estado estão mencionados no texto estruturado. A figura 5 apresenta o diagrama de rede obtido.

O amplo entendimento obtido possibilitou críticas ao desenho do sistema. Por exemplo. o processo de validação não era o responsável por todos os procedimentos relacionados

à esta função; a função de edição não pertencia ao processo editor. Estas críticas, embora não registradas na metodologia JSD, foram utilizadas como fonte de referência para orientar o re-desenho do sistema.

4.2 Re-Desenho do HyperLex

Na construção do HyperLex seguiu-se a metodologia JSD na sua sequência natural. A grande diferença foi a reutilização de conhecimento obtido previamente tanto na tarefa de recuperação como no uso do Hyperties [Leite 90]. Este conhecimento era composto de informações registradas e informações não registradas. As informações registradas estavam nos artefatos JSD (Figura 3, 4 e 5), ou em textos livres contendo as críticas. As não registradas ficaram circunscritas aos autores.

A primeira parte da construção do HyperLex levantou as entidades e ações. No caso das entidades coincidentes com processos recuperados, a estruturação dessas entidades pela sequência das ações (diagrama estrutura/entidade) foi facilitada. Em alguns desses casos, alterações na sequência foram efetuadas em consequência de críticas.

O modelo inicial foi feito com base nos diagramas estrutura/entidade. Nessa fase não foi diretamente utilizado os conhecimentos oriundos da recuperação.

Na fase de função foi onde efetivamente se aplicou o re-desenho. Neste momento foram reutilizados os artefatos recuperados e se incorporou a nova funcionalidade (Figura 1). O aumento de funcionalidade exigiu a criação de uma nova interface. Esta interface está distribuída em cada um dos processos, ou seja, essas interfaces foram especificadas como **funções implícitas**[5]. Uma vez composto o diagrama de rede foram escritos os textos estruturados correspondentes, reutilizando os textos recuperados e as críticas.

Com relação a interface é importante ressaltar que muitas das idéias surgiram dada a vivência que se tinha com o problema. Tanto pela utilização do Hyperties na criação do léxico da biblioteca, como também pela utilização do editor de léxicos.

De posse do texto estruturado recuperado e do diagrama de rede recuperado foi elaborado o diagrama de implementação (Figura 6). O próximo passo foi o detalhamento dos textos estruturados para as linguagens de implementação, C e Sunview. Convém salientar que o Makefile do HyperLex é cópia fiel do diagrama de implementação. No detalhamento dos textos estruturados muitas vezes se utilizou uma codificação direta do programa original em Pascal para a linguagem C. Este processo foi obviamente possível porque havia um elo entre o programa original e o novo programa, isto é, o texto estruturado. Um exemplo é a codificação de `Alterar Entrada` onde o texto estruturado do `Léxico` foi utilizado. Nas Figuras 4, 7 e 8 estão mostrados: o texto estruturado, o programa original em Pascal e o programa em C.

O HyperLex foi implementado no ambiente Unix/SUN usando a linguagem C e funções do SunView. O sistema encontra-se operacional e está sendo utilizando para a confecção de Léxicos Ampliados da Linguagem.

[5] JSD embedded functions.

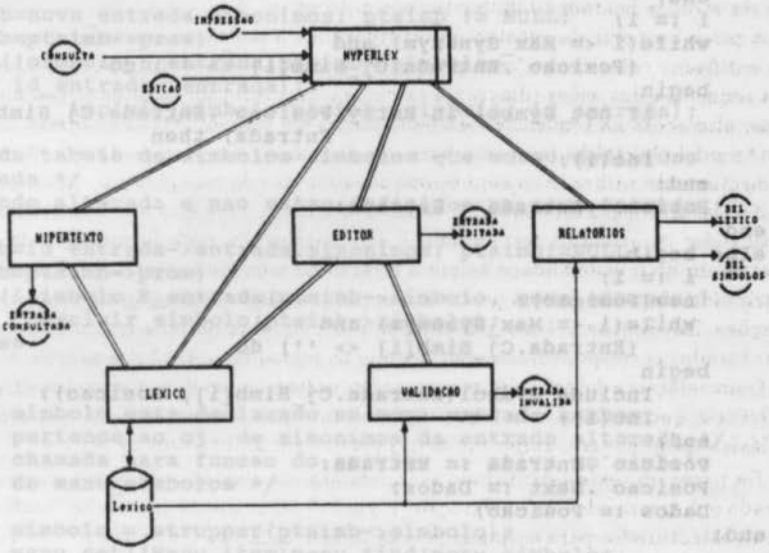

Figura 6: Diagrama de Implementação do HyperLex

```
Procedure Dictionary,Alterar_Entrada(Posicao : Position;
                                     Entrada : Enter);
Var i : byte;
begin
    if Posicao <> NullPosition then
    begin
        i := 1;
        while(i <= Max_Synonym) and (Entrada.Cj_Simb[i] <> '') do
        begin
            if not Symbol_in_Entry(Entrada.Cj_Simb[i],
                                   Posicao^.Entrada) then
                Include_Symbol(Entrada.Cj_Simb[i], Posicao);
            Inc(i);
        end;
        i := 1;
        while(i <= Max_Synonym) and
              (Posicao^.Entrada.Cj_Simb[i] <> '') do
        begin
            if not Symbol_in_Entry(Posicao^.Entrada.Cj_Simb[i],
                                   Entrada) then
            Inc(i);
        end;
        Posicao^.Entrada = Entrada;
    end
    else begin
        i := 1;
        new(Posicao);
        while(i <= Max_Synonym) and
              (Entrada.Cj_Simb[i] <> '') do
        begin
            Include_Symbol(Entrada.Cj_Simb[i], Posicao);
            Inc(i);
        end;
        Posicao^.Entrada := Entrada;
        Posicao^.Next := Dados;
        Dados := Posicao;
    end;
end;
```

Figura 7: Codigo Pascal

```
alterar_entrada(id_entrada, nova_entrada)
_PT_ENTRADA     id_entrada;
_ENTRADA        nova_entrada;
{
    _SIMBOLO      simbolo; /* conversao caixa alta do simbolo a
                              serem ativados no menu_simbolos */
    _Cj_SIMBOLOS *ptsimb; /* ponteiro auxiliar p/ inclusao e
                             exclusao de simbolos na tabela */

/* inclui na tabela de simbolo os simbolos que estao definidos
   na nova */
/* entrada e nao estao definidos na entrada que esta sendo
   alterada */

for(ptsimb=nova_entrada.sinonimos; ptsimb != NULL;
   ptsimb=ptsimb->prox)
      if(!simbolo_E_entrada(ptsimb->simbolo,
         id_entrada->entrada))
            incluir_simbolo(ptsimb->simbolo, id_entrada);

/*exclui da tabela de simbolos simbolos que estao definidos
  na entrada */
/*esta sendo alterada e nao estao definidos na nova entrada */

for(ptsimb=id_entrada->entrada.sinonimos; ptsimb!=NULL;
   ptsimb=ptsimb->prox)
      if(!simbolo_E_entrada(ptsimb->simbolo, nova_entrada))
          excluir_simbolo(ptsimb->simbolo);
      else
      {
      /* simbolo esta declarado na nova entrada tambem
         pertence ao cj. de sinonimos da entrada alterada */
      /* chamada para funcao do surview -> ativa um simbolo
         do menu_simbolos */

         simbolo = strupper(ptsimb->simbolo);
         menu_set((Menu_item)menu_find(menu_simbolos,
                                       MENU_STRING,simbolo,0),
                                       MENU_INACTIVE, FALSE,
                  0);
         free(simbolo);
      }

 /* libera o conteudo da entrada alterada */
   apagar_conteudo(id_entrada);

 /* armazena o novo conteudo da entrada */
   id_entrada->entrada=nova_entrada;
   return(X);
}
```

Figura 8: Codigo C

5 Conclusões e Trabalhos Futuros

Neste artigo procuramos apresentar uma proposta diferente de reutilização. Esta proposta foi materializada numa experiência real, onde foi feito um trabalho de re-engenharia que envolveu a recuperação e o re-desenho de um sistema de hipertexto para apoio a produção de linguagens da aplicação. A metodologia JSD foi usada tanto na recuperação da estrutura e do comportamento do sistema, como também no novo projeto. A reutilização não foi feita a nível de código, mas sim à nível de desenho.

Na execução do presente estudo de caso detalhamos várias estratégias usadas em [Leite 91]. Neste detalhamento encontramos vários problemas, que foram encaminhados segundo o desenvolvimento da recuperação do editor de léxicos. Dentre os problemas encontrados, um dos que mais gerou discussão foi o referente a vetores de estado e fila de mensagens. A princípio todas as ligações foram interpretadas como passagem de parâmetros, o que caracteriza a ligação por dutos no diagrama de implementação, e portanto não apresentariam ligações por vetores de estado. Essa caractéristica foi confirmada pela ausência de áreas comuns. Apesar disso, na primeira versão desse artigo a Figura 3 apresentava um arquivo ligado somente ao processo Léxico. Este arquivo não se caracterizava como um vetor de estados, visto que somente o processo Léxico tinha acesso a ele. Na versão atual omitimos este arquivo. Outro problema encontrado foi a transformação (manual) do código recuperado em texto estruturado, em razão da complexidade das estruturas de controle utilizadas no código original. É importante ressaltar que apesar de recuperarmos o diagrama de rede, não conseguimos estabelecer heurísticas para diferenciar entre processos do tipo função e processos do tipo entidade.

Com o objetivo de proceder manutenções, o engenheiro de software precisa entender o sistema em questão. Biggerstaff [Biggerstaff 89] lista as seguintes questões relacionadas com esta tarefa:

1. Quais são os modulos?
2. Quais os ítens de dados mais importantes?
3. Quais são os artefatos utilizados para registrar a recuperação?
4. Quais são as abstrações informais utilizadas?
5. Qual é a relação das abstrações de desenho com o código.

Nossa estratégia responde as questões 1 e 2 com o artefato, não constante da metodologia JSD, chamado índice (Figura 2). A questão 3 refere-se à visão do processo de software, no nosso caso utilizamos a metodologia JSD aplicada em ordem inversa. A questão 4, no nosso caso, diz respeito as críticas, isto é o resultado da análise do desenho recuperado, e ao conhecimento dos autores a respeito do problema. A questão 5 é respondida pelo processo utilizado, isto é os artefatos JSD têm relações pré-definidas e que são explicitadas na recuperação do desenho, Figuras 4, 7 e 8.

Diferentemente de Biggerstaff, sistema Desire [Biggerstaff 89], não propomos, neste artigo, uma arquitetura para a automatização do processo de recuperação. Esta proposta, no entanto, está nos nossos planos [Lucena 91]. Em especial estamos interessados em integrar ao sistema JSD/PUC [Prado 91] estas facilidades de recuperação, levando em consideração, principalmente, os aspectos do registro de decisões e justificativas de

desenho. No caso da recuperação, um dos aspectos que precisam ser mais bem estudados é como utilizar o processo de registro de decisões e justificativas como forma de registro do que aqui chamamos genericamente de críticas.

As experiências obtidas com o caso apresentado e com a recuperação de Draco são subsídios importantes para uma arquitetura orientada à assistência ao processo de re-engenharia. No entanto, é importante frizar que nossos resultados são parciais e oriundos de dois estudos de caso. Um dos esforços futuros de pesquisa é justamente utilizar esta estratégia em outros casos, de forma a obter mais dados para a validação da prática de re-engenharia aqui proposta. É importante salientar, também, que várias informações utilizadas e produzidas durante o processo de recuperação e re-engenharia precisam ser sistematizadas.

Referências

[Ambrosio 89] Ambrosio, A. e Velasco, R. *Um Sistema para Execução de Especificações JSD*, III Simpósio Brasileiro de Engenharia de Software, SBC, 1989.

[Baxter 90] Baxter, I. *Transformational Maintance by Reuse of Design Histories*. Tese de Doutorado, University of California, Irvine, USA; Nov., 1990.

[Biggerstaff 89] Biggerstaff, T. *Design Recovery for Maintance and Reuse*, IEEE Computer, 22(7); Jul., 1989; pags. 36-49.

[Cameron 86] Cameron, J. *An Overview of JSD*, IEEE Transaction on Software Engineering, vol SE-12, no. 2; Fev., 1986, pags. 222-240.

[Chikofsky 90] Chikofsky, E. e Cross II, J. *Reverse Engineering and Design Recovery: A Tazonomy*, IEEE Software; Jan., 1990; pags. 222-240.

[Jackson 83] Jackson, M. *System Development*, Prentice-Hall International; 1983.

[Leite 89] Leite, J.C.S.P. Elicitation of Application Languages. *In Monografias em Ciência da Computação, PUC-RIO*, no. 30. 1989.

[Leite 90] Leite, J.C.S.P. e Franco, A. *O Uso de Hipertexto na Elicitação de Linguagens da Aplicação*. IV Simpósio Brasileiro de Engenharia de Software, SBC, 1990.

[Leite 91] Leite, J.C.S.P. e Prado, A.F. *Design Recovery - A Multi-Paradigm Approach*, First International Workshop on Software Reusability, Dormund, Germany; Jul., 1991.

[Lucena 91] Lucena, C.J.P.; Leite, J.C.S.P.; Schwabe, D.; Fuks, H.; A Research Agenda on Software Design, *Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ*, March, 1991.

[Maciel 90] Maciel, G., Costa, J. e Baccar, J. *Editor de Léxicos. Trabalho Final do Curso de PSS*, Departamento de Informática - PUC-RIO, Jul, 1990.

[Parikh 88] Parikh, G. *Technics of Program and System Maintance* (2a. edição), QED Information Sciences, Inc., 1988.