CRITÉRIOS PARA A AVALIAÇÃO DA COESÃO

Arndt von Staa
(email: arndt@inf.puc-rio.br)

PUCRioInf - MCC 26/93

ABSTRACT

A set of quality control criteria is presented. These criteria evaluate the quality factor *cohesion*. The criteria may be used whenever representations having several levels of abstraction are built. The criteria may be used with respect to several representation languages, not just with respect to structured design. When working with elements at different levels of abstraction a solution of each abstract element must be provided. This solution is defined at a lower level of abstraction. Furthermore, this solution must satisfy several relations with respect to the abstract element it intends to solve. The solution must also satisfy several structural properties. The objective of the quality control criteria is to provide a means to objectively decide wheter a given solution is good or not.

KEYWORDS: Cohesion, Decomposition, Levels of abstraction, Quality criteria, Quality factors

RESUMO

É apresentado um conjunto de critérios de avaliação para o fator de qualidade *coesão*. Este conjunto de critérios se aplica em todas as ocasiões em que se produz representações com vários níveis de abstração. O conjunto destina-se ao uso em diversas linguagens de representação, não somente ao gerar projetos estruturados. Ao utilizar níveis de abstração, precisa-se, para cada elemento abstrato, definir uma solução. Esta solução precisa guardar diversas relações com o elemento abstrato que pretende resolver, além de satisfazer diversas propriedades estruturais. O objetivo do conjunto de critérios é determinar se uma dada solução é boa ou não.

PALAVRAS CHAVE: Coesão, Critérios de qualidade, Decomposição, Fatores de qualidade, Níveis de abstração

1. Introdução

Desde que se projeta programas existe a preocupação com critérios de avaliação da estrurtura de seus projetos. Na literatura atual são propostos diversos critérios, variando desde propostas pouco razoáveis tais como: “um módulo deve ter menos do que 55 linhas”, até propostas tais como: “um módulo deve ter elevada coesão e baixo acoplamento” [Yourdon79]. Infelizmente, a maioria destas definições é excessivamente subjetiva para servir como um instrumento eficaz de avaliação da qualidade de um projeto.

Neste artigo propomos um conjunto de critérios de avaliação para o fator de qualidade *coesão*, bem como uma generalização do uso deste conceito em diferentes linguagens de representação. Na seção 2 caracterizaremos abstrações e conjuntos de solução como instrumentos básicos de projeto de sistemas de programação. Na seção 3 abordaremos, resumidamente, alguns dos conceitos relacionados com especificação e controle de qualidade. Na seção 4 apresentaremos os critérios de avaliação que constituem o fator de controle de qualidade coesão. Finalmente, na seção 5 avaliaremos os conceitos introduzidos.

2. Conjuntos de solução

Ao desenvolvermos sistemas de programação operamos continuamente com *abstrações* e com *decomposições* destas abstrações. Por exemplo, ao projetarmos um determinado programa, lançamos mão de *estruturas de chamadas de funções* (estruturas modulares [Yourdon79], estruturas de pacotes [STAA92]) e de estruturas organizacionais (decomposição sucessiva, projeto estruturado). Nestas estruturas cada componente abstrato é descrito através de um conjunto de outros componentes. Estes, por sua vez podem ser abstrações em um nível de abstração menor ou, então, ser implementações acabadas (componentes concretos).

De modo semelhante, ao projetarmos estruturas de dados complexas, é comum lançarmos mão de módulos (objetos, classes, tipos abstratos). Estes módulos tornam disponíveis os tipos de dados e as operações capazes de implementar a abstração de dados desejada, encapsulando os detalhes de sua implementação. O conjunto de elementos tornados disponíveis na interface de um módulo é uma solução da abstração que este módulo representa. Por exemplo, ao definirmos o módulo *tabela de símbolos* temos em mente um determinado conceito. Este, por sua vez, se realiza através das operações *inserir símbolo*, *obter nome do símbolo*, *verificar se símbolo existe*, etc., e dos tipos de dados *identificação do símbolo*, *nome de símbolo* etc.

Concluindo, quando especificamos, projetamos ou implementamos sistemas de programação, trabalhamos com *abstrações raiz* e com conjuntos de elementos que resolvem ou implementam estas abstrações, os *conjuntos de solução*. A figuras 1 e 2 ilustram estas definições.

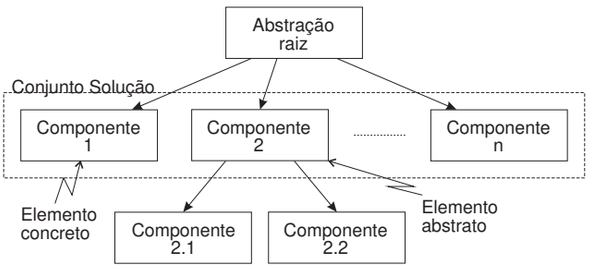

Figura 1. Ilustração das definições de abstração raiz, conjunto de solução, elemento de solução concreto, elemento de solução abstrato.

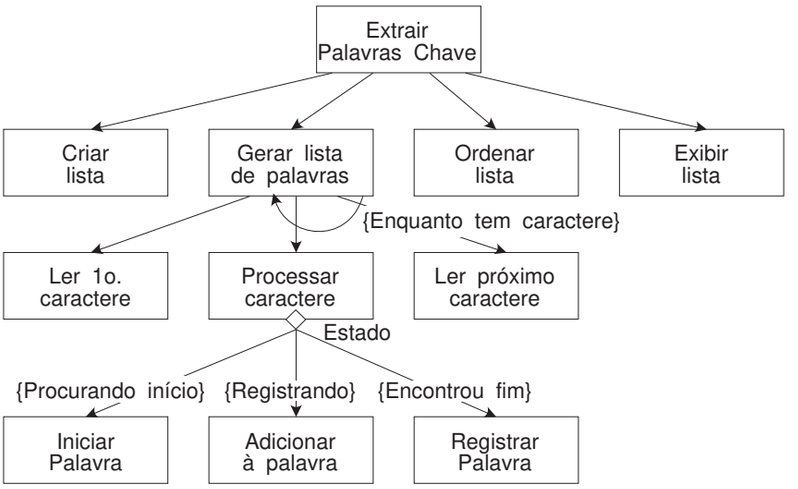

Figura 2. Estrutura de solução correspondente a código de programa.

Conjuntos de solução são compostos por diversos elementos *concretos* ou *abstratos*. Elementos concretos correspondem a soluções acabadas, com relação a eles nada mais precisará ser feito. Elementos abstratos requerem a criação de outros conjuntos de solução e são resolvidos através destes. Desta forma criam-se *estruturas de solução* tão profundas quanto se queira.

Por exemplo, em um fluxo de dados o processo "Operar conta corrente" pode ser decomposto em outros processos "Abrir conta corrente", "Movimentar conta corrente" e "Fechar conta corrente". De modo semelhante, ao procurarmos uma solução para o tipo abstrato de dados "Pilha de inteiros", podemos gerar as operações "Criar pilha vazia", "Empilhar inteiro", "Obter inteiro do topo", "Desempilhar", "Verificar se a pilha está vazia" e "Destruir pilha". Finalmente, se a solução de um determinado problema pode ser alcançada por aproximações sucessivas, podemos imaginar um conjunto de solução contendo os seguintes elementos: "Validar dados de entrada", "Calcular a primeira aproximação", "Calcular sucessivas aproximações".

Os elementos de um conjunto de solução não precisam ser disjuntos. Dito de outra forma, é perfeitamente legal que diferentes conjuntos de solução possuam interseções não vazias. Cabe salientar que elementos comuns a vários conjuntos solução são elementos reutilizados. É legal, ainda, que um elemento de um conjunto de solução seja um antecessor deste conjunto, formando uma estrutura recursiva. A figura 3 ilustra estruturas de solução.

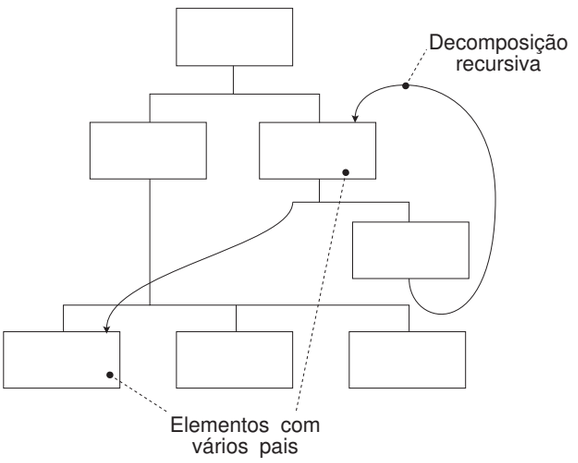

Figura 3. Ilustração de uma estrutura de solução genérica

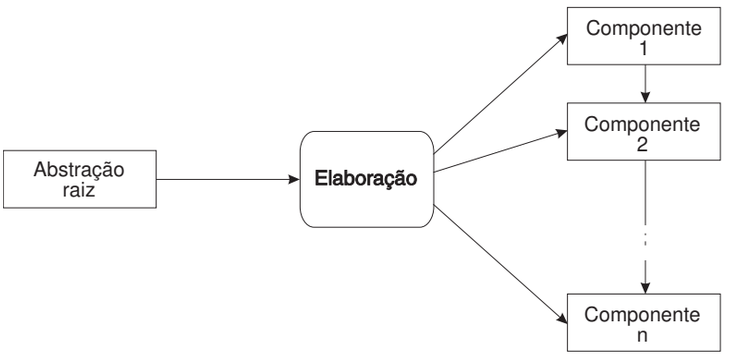

Figura 4. Passo de projeto e conjuntos solução

Cada conjunto de solução surge como conseqüência de um passo de projeto, ver figura 4. Passos de projeto podem ser de *decomposição* ("top down") ou de *agregação* ("bottom up"). De maneira geral, é irrelevante saber se criamos os conjuntos de solução através da decomposição de uma abstração raiz, ou se procuramos uma abstração raiz para um conjunto de elementos já existentes. O que é relevante, é o fato que trabalhamos com uma abstração raiz e um conjunto de solução desta abstração.

Passos de projeto podem ser *elementares* ou *compostos*. Um passo de projeto elementar é, usualmente, realizado por uma única pessoa. Passos de projeto compostos são realizados através de uma rede ou uma seqüência de passos de projeto de nível de abstração menor, à semelhança da decomposição de uma abstração raiz em um conjunto de solução. A sucessão de passos de projeto formam o *processo de desenvolvimento*. Trabalho em equipe é possível se a *estrutura de passos de projeto* admite que diversos passos de projeto sejam realizados simultanemente.

A conseqüência imediata dos conceitos introduzidos nesta seção, é a necessidade do conjunto de solução corresponder a uma solução exata da abstração raiz. Entretanto, não basta a solução ser *correta* e *completa*, é necessário ser uma *boa solução*. Em adição, as representações estruturadas produzidas devem facilitar eventuais alterações a serem realizadas durante a manutenção do sistema de programação. A qualidade do conjunto de solução é avaliada pelo fator de qualidade **coesão**.

3. Especificação e controle de qualidade

Para termos certeza que vamos construir um objeto com uma determinada qualidade, é óbvio que precisamos saber de ante-mão quais são as propriedades que determinam esta qualidade. Precisamos, então, saber definir e avaliar ou medir propriedades determinates da qualidade. Estas propriedades dividem-se em dois grandes grupos:

1- **qualidade de serviço**, descreve as propriedades que o software deve possuir para que o *serviço por ele produzido seja aceitável*. A qualidade de serviço é definida através da **especificação essencial*** do software como um todo, bem como dos componentes que o cosntituem. Cada elemento de cada conjunto de solução possui uma especificação essencial. Esta define precisamente o que este elemento deve realizar, sem se preocupar com a forma da realização.

2- **qualidade de engenharia**, descreve as propriedades que a implementação do software deve ter para que *represente uma solução aceitável*. A qualidade de engenharia

---

* O termo *especificação essencial* é costumeiramente utilizado em *análise essencial de sistemas*, ver [McMenamin84][Maffeo92]). Tomamos este termo emprestado, uma vez que, conceitualmente, a nossa definição é exatamente a mesma que a encontrada na literatura, embora a forma de representar seja diferente, como são diferentes, também, as etapas em que é produzida.

é definida através da **especificação de projeto** . Conseqüentemente, está diretamente relacionada com a organização e composição dos conjuntos de solução.

Enquanto estivermos produzindo ou mantendo especificações essenciais e projetos, dispomos de documentos (*representações*). Esses, em geral, não permitem a observação direta do comportamento do componente sendo especificado ou projetado. Mesmo assim temos que controlar a qualidade destas representações. Isto é realizado através da avaliação das representações segundo determinados *critérios de qualidade*. Critérios de qualidade estabelecem mais do que meros padrões de avaliação da qualidade. A existência da definição e cobrança desses critérios induz os desenvolvedores a procurarem atingir as metas de qualidade estabelecidas pelo critério. Critérios de qualidade podem ser agregados, formando *fatores de qualidade*. Um destes fatores é a **coesão**, o objeto de estudo deste artigo.

A avaliação da coesão se dá *passo a passo* durante o processo de desenvolvimento. A cada vez que uma estrutura formada por uma abstração raiz e respectivo conjunto de solução for gerada ou alterada, esta estrutura será avaliada. Reduzem-se, assim, as ocasiões em que são desenvolvidos elementos a partir de especificações incorretas. De nada adianta continuar detalhando ou desenvolvendo, quando já se sabe que a solução proposta é inadequada.

Como resposta a uma deficiência encontrada durante a avaliação, pode-se:

1- ou modificar a composição do conjunto de solução.

2- ou rever a especificação da abstração raiz.

3- ou uma combinação destas duas.

Cabe salientar que a revisão da especificação de uma abstração raiz, implica na revisão de todos os conjuntos solução de que esta abstração raiz faça parte. Isto, por sua vez, evidencia que o processo de desenvolvimento de uma estrutura de soluções raramente será estritamente "top down". Na realidade, o processo de desenvolvimento tende a ser um processo de *amadurecimento.* Neste, na medida que se vai adicionando conjuntos de solução e/ou revendo especificações, partes da estrutura de solução se tornam mais estáveis, enquanto que outras partes da estrutura de solução ainda são sujeitas a um grande volume de alteração, mesmo que estas não correspondam a folhas.

4. Critérios de avaliação da coesão

Uma das primeiras referências a coesão, ainda sob o nome de *força* ("module strength"), encontra-se em [Myers72]. Nesta são avaliadas exclusivamente as propriedades de projetos compostos por diversos pacotes (funções, subrotinas, "procedures", etc.). Apesar do conceito coesão ter surgido há algum tempo, a definição encontrada na literatura não possui as características necessárias que permitam utilizá-lo como um instrumento eficaz de especificação e controle de qualidade. As definições de coesão encontradas na literatura [Page-Jones80][Yourdon79][DeMarco78][Gane79][Rocha83] são, de maneira geral, informais ou ambíguas. As definições são apresentadas, tipicamente, sob a forma de listas de caracterizações da coesão, semelhantes à lista [Staa83] que ilustramos, parcialmente, a seguir:

incidental: não existe relacionamento entre os elementos do conjunto de solução;

lógica: cada elemento efetua uma função de um conjunto de funções correlatas, não existindo função ou noção única definida por este conjunto.

temporal: existe somente relação de ocasião de execução entre os elementos do conjunto de solução.

funcional: o conjunto implementa exatamente uma única ação de transformação ou de controle de dados.

Esta forma de definir a coesão apresenta várias deficiências, entre elas:

1- é *imprecisa*, freqüentemente o avaliador fica em dúvida quanto à "nota" que deverá escolher;

2- é *individualista*, diferentes autores publicam listas de "valores" diferentes, todas elas baseadas em arrazoados subjetivos e de ordem pessoal;

3- é *limitada*, avalia somente o grau de modularidade (empacotamento) de um projeto de programa.

Estas limitações são conseqüência direta do modo de definir a coesão. Precisamos de uma definição que permita avaliar a estrutura da solução para diversas classes de abstrações, tais como hierarquias de fluxos de dados, hierarquias de dicionários de dados, hierarquias de máquinas de estados, organização de textos, etc., além, é claro, de projetos estruturados de pacotes, tipos, código e dados. Sobretudo, a definição deve levar a avaliações confiáveis e independentes do avaliador humano.

Têm surgido na literatura outras referências sobre os temas coesão e projeto modular. Por exemplo [Rising92] define métricas para a avaliação da coesão e do acoplamento em pacotes Ada. Este mesmo artigo faz uma pequema resenha de propostas de métricas feitas por outros autores. No entanto, a idéia básica é mantida, ou seja, são dadas diversas definições com caracterizações razoavelmente precisas. A crítica apresentada acima continua valendo. Cabe ressaltar que sendo satisfeitos os critérios descritos a seguir, atinge-se o nível máximo de coesão também segundo o que está definido no artigo referenciado.

Seguindo o modelo de avaliação da qualidade introduzido em [Rocha83], definimos coesão como um *fator de qualidade* composto pelos seguintes *critérios de avaliação*:

Necessidade: cada um dos elementos contidos no conjunto de solução contribui para resolver a abstração raiz.

Suficiência: os elementos do conjunto solução resolvem completamente a abstração raiz.

Integrabilidade: é possível compor a implementação da abstração raiz a partir das implementações de cada um dos elementos do conjunto de solução, desde que estes estejam em conformidade com as suas especificações essenciais.

Minimalidade: não existe sub-conjunto do conjunto de solução, para o qual seja possível definir uma abstração raiz própria.

Ortogonalidade: cada elemento do conjunto de solução resolve uma parte da abstração raiz e que não é resolvida por qualquer outro componente do conjunto de solução. Idealmente, os elementos do conjunto de solução deveriam ser *estritamente ortogonais*, no sentido que cada um possua nenhuma superposição com qualquer um dos demais elementos do conjunto.

Completeza de interface: a especificação essencial de cada elemento do conjunto de solução está completa, assegurando que basta conhecer esta especificação para construir uma implementação correta e integrável para o elemento.

Viabilidade: cada elemento do conjunto de solução, ou já é uma solução concreta, ou, em princípio, admite uma solução.

Nas seções a seguir discutiremos em mais profundidade cada um dos critérios acima identificados.

4.1. Descrição do exemplo

Um *módulo* é um componente de programa independentemente compilável. Utilizamos aqui uma definição para módulo semelhante à encontrada em Modula2 [Wirth82][*]. Um módulo é composto por diversos *elementos*. Os elementos podem ser: pacotes

---

* Note que em [Yourdon79] um módulo corresponde, em linhas gerais, a uma função ou “procedure”.

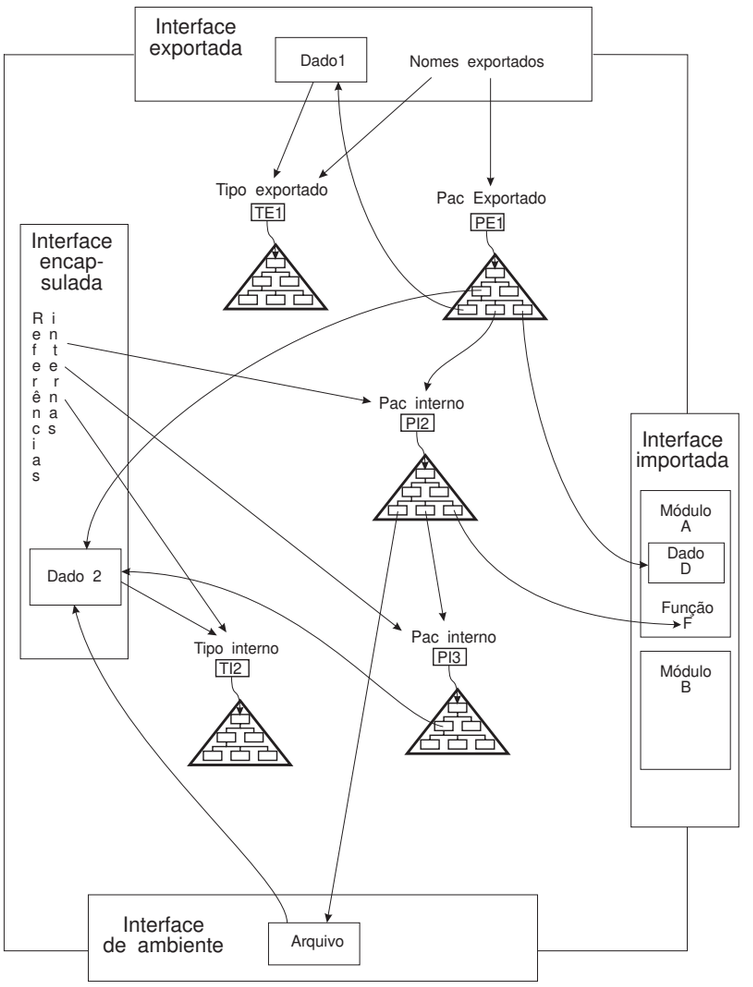

Figura 5. Ilustração da composição de um módulo.

(funções, "procedures", subrotinas, etc.), diversos tipos ("records", "structs", "typedefs", estruturas COBOL, blocos COMMON, etc.) e dados encapsulados, a *interface encapsulada.* Através da *interface exportada*, módulos tornam disponíveis pacotes, tipos e dados para que possam ser utilizados por outros módulos. Através da *interface importada*, módulos (*módulos cliente*), utilizam os elementos exportados por outros módulos (*módulos servidores*). Os elementos encapsulados e os elementos *públicos* contidos nas interfaces exportadas e importadas servem como mecanismo de comunicação entre os elementos internos do módulo. A figura 5 ilustra a composição de um módulo.

Para ilustrar os critérios de avaliação do fator coesão, vamos utilizar um projeto de um módulo que implementa um sistema de arquivos. Vamos assumir que o **módulo sistema de arquivos** tenha sido projetado, inicialmente, através do seguinte conjunto de solução:

```
Abrir arquivo para leitura
Abrir arquivo para gravação
Ler registro
Gravar registro
Imprimir registro
Posicionar registro
```

4.2. Critério necessidade

4.2.1. Definição

Cada um dos elementos contidos no conjunto de solução contribui para resolver a abstração raiz. Ou seja, não existe elemento pertencente ao conjunto de solução que possa ser retirado, ou reduzido em sua funcionalidade, sem comprometer a capacidade deste elemento resolver a sua parte da abstração raiz.

4.2.2. Exemplo

O pseudo-código a seguir possui elementos desnecessários:

```
calcular imposto devido
   obter renda líquida
   obter renda não tributável
   obter alíquota e desconto
```

Como *renda não tributável* não entra no cálculo do imposto devido, este elemento pode ser retirado sem comprometer a solução da abstração raiz "calcular imposto devido".

Consideremos agora o módulo sistema de arquivo. Neste módulo está definido o pacote *imprimir registro*. Sistemas de arquivos devem esconder do usuário as particularidades dos meios físicos em que são registrados os arquivos. Assim, escrever em um disco ou em uma impressora é a mesma operação do ponto de vista do sistema de arquivos. Isso torna desnecessário o pacote *imprimir registro*.

4.2.3. Relevância

Elementos desnecessários somente contribuem para aumentar os custos de projeto, desenvolvimento, controle de qualidade e manutenção, além de confundir quem venha a ler a documentação.

4.2.4. Como realizar a avaliação

Examina-se cada elemento do conjunto de solução, procurando determinar:

1- a parcela da abstração raiz que é resolvida pelo elemento.

2- se existe alguma parcela no elemento sendo examinado que não está diretamente comprometido com a solução da abstração raiz.

4.3. Critério suficiência

4.3.1. Definição

Os elementos do conjunto solução resolvem completamente a abstração raiz. Ou seja, não existe nenhuma parte da abstração raiz que não esteja sendo resolvida por algum dos elementos do conjunto de solução. Tampouco, se conhece um componente que se possa adicionar ao conjunto solução de modo que se tenha uma solução mais completa.

4.3.2. Exemplo

O pseudo-código a seguir é insuficiente

```
calcular imposto devido
   obter renda líquida
   obter alíquota e desconto
```

Tal como escrito, os trecho de código somente obtem dados sem fazer cálculo algum com eles. Ou seja, falta uma operação do gênero *efetuar o cálculo do imposto devido*.

O módulo sistema de arquivos utiliza os conceitos: *arquivo* (um espaço de dados em algum meio, por exemplo disco), *arquivo em uso* (um indicador que determinado programa está utilizando determinado arquivo) e *registro* (um espaço de dado contido no

arquivo). Estes três conceitos, correspondem a tipos e são necessários, embora não estejam na definição original. Também não se encontra na definição original o pacote *destruir arquivo*, apesar de arquivos poderem ser criados ao serem abertos para a gravação.

4.3.3. Relevância

Especificações e projetos incompletos são uma fonte de "manutenção de instalação", cara e desgastante. Soluções incompletas levam muitas vezes a escolhas de algoritmos ou de estruturas de dados inapropriados quando se considera o conjunto completo.

4.3.4. Como realizar a avaliação

Procura-se identificar se existem partes da abstração raiz que não estejam sendo resolvidas por algum dos elementos do conjunto de solução. Procura-se verificar, ainda, se é possível encontrar um componente que se possa adicionar ao conjunto solução de modo que se tenha uma solução mais completa.

4.4. Critério integrabilidade

4.4.1. Definição

É possível compor a implementação da abstração raiz a partir das implementações de cada um dos elementos do conjunto de solução, desde que estes estejam em conformidade com as suas especificações essenciais. Ou seja, cada elemento do conjunto solução será desenvolvido isoladamente. Uma vez todos eles resolvidos, eles serão integrados. Esta integração não deve requerer a revisão de qualquer um dos componentes, tampouco, deve requerer a incorporação de adaptadores de interface (também conhecidos por *bacalhaus*).

4.4.2. Exemplo

No módulo sistema de arquivos temos que definir uma sintaxe para o nome simbólico de arquivo. Temos, que escolher uma estrutura de dados para registrar os arquivos conhecidos, também conhecida por *diretório*. Temos que escolher como uma estrutura de dados para arquivo em uso, também conhecido por *file control block*. Finalmente, temos que escolher uma estrutura de dados para designar o acesso a um registro de um arquivo em uso. Analisando um pouco mais, veremos que temos ainda que definir condições de retorno tais como: *fim de arquivo*, *erro de leitura*, *erro de gravação*, *erro de abertura*, *operação OK*, etc. As estruturas de dados diretório, arquivo em uso e acesso a registro são estruturas internas e devem ser encapsuladas. Embora encapsuladas, a sua existência é pública uma vez que estas estruturas determinam os estados utilizados na especificação formal. O que é encapsulado é na realidade somente a organização física. Já o nome de arquivo, a identificação de arquivo em uso e a condição de retorno são estruturas externas e devem ser exportadas.

A integração das diversas operações se dá através destas estruturas de dados. A escolha destas estruturas deve ser tal que se possa atingir todos os requisitos do módulo, como, por exemplo, independência de dispositivo físico, controle de acesso simultâneo a partir de diversos programas, encapsulamento das estruturas de dados internas, etc.

4.4.3. Relevância

Problemas de integração são insistentemente citados como sendo os problemas mais freqüentes e mais complexos no desenvolvimento e na manutenção de software. Assegurando-se integrabilidade, eliminam-se focos de problemas de integração.

4.4.4. Como realizar a avaliação

O conjunto de solução sendo coeso, implica na existência de interfaces entre cada par de componentes deste conjunto. Estas interfaces devem estar explicitamente definidas. Além disso, a interface com que o elemento A vê o elemento B tem que ser a mesma interface com que o elemento B vê o elemento A. Para os elementos cuja existência é pública, mas

cuja implementação é encapsulada, as especificações públicas estabelecem os requisitos a serem satisfeitos pela implementação, qualquer que seja esta.

4.5. Critério minimalidade

4.5.1. Definição

Não existe sub-conjunto do conjunto de solução, para o qual seja possível definir uma nova abstração raiz. A substituição do sub-conjunto pela abstração raiz identificada reduz a cardinalidade dos conjuntos de solução e, conseqüentemente, aumenta o grau de hirearquia da solução final dada ao problema. Caso satisfaça este critério, o conjunto é minimal, uma vez que não é possível reduzí-lo através da substituição de um sub-conjunto por um único elemento.

4.5.2. Exemplo

Na estrutura de solução:

```
operar conta corrente
    abrir conta corrente
    depositar em conta corrente
    retirar de conta corrente
    consultar conta corrente
    fechar conta corrente
```

as operações "depositar", "retirar" e "consultar" podem ser agregadas em "movimentar conta corrente", levando à estrutura mais profunda:

```
operar conta corrente
    abrir conta corrente
    movimentar conta corrente
        depositar em conta corrente
        retirar de conta corrente
        consultar conta corrente
    fechar conta corrente
```

Conforme já foi identificado acima, no módulo sistema de arquivos estamos tratando dos seguintes três módulos:

```
Gerenciar arquivos, composto por
    Tipos de dados
        Nome de arquivo
        Condição de retorno
        Valor de arquivos
    Pacotes
        Criar arquivo
        Verificar se arquivo existe
        Destruir arquivo

Gerenciar arquivo em uso, composto por
    Tipos de dados
        Identificação de arquivo em uso
        Descritor de arquivo em uso
    Pacotes
        Verificar se arquivo está em uso
        Abrir arquivo para leitura
        Abrir arquivo para gravação
        Fechar arquivo

Acessar registros de arquivo, composto por
    Tipos de dados
        Descritor de acesso a registro
    Pacotes
        Verificar se registro existe
        Posicionar registro
        Ler registro
        Gravar registro
```

4.5.3. Relevância

A redução da cardinalidade dos conjuntos de solução facilita o entendimento, a avaliação da correção e a manutenção destes conjuntos. Só isso já implica em querer-se trabalhar com conjuntos de solução minimais. Além disso, a possibilidade de fatorar um conjunto de solução é um indicador de que a interdependência entre os elementos do sub-conjunto fatorado e do restante do conjunto de solução tende a ser pequena. No exemplo do sistema de arquivos é fácil observar que a interdependência entre *ler registro* e *destruir arquivo* é muito pequena, já entre *posicionar registro* e *ler registro* é muito grande. Utilizando a definição de coesão encontrada na literatura convencional, pode-se observar que quanto mais próximo de uma cardinalidade minimal estiver o conjunto de solução, mais forte será a coesão e, inversamente, quanto mais afastado estiver, mais fraca será a coesão.

4.5.4. Como realizar a avaliação

Procura-se sub-conjuntos formados por dois ou mais elementos e que possam representar um conceito próprio. Sendo possível encontrar um sub-conjunto, substitui-se esse sub-conjunto pela correspondente abstração raiz. O sub-conjunto passa a ser o conjunto solução desta nova raiz.

4.5.5. Uma forma alternativa de avaliar a minimalidade

Ao invés de laboriosamente procurar um sub-conjunto, pode-se examinar se todos os elementos do conjunto de solução estão no mesmo nível de abstração. Caso não estejam, separa-se os elementos, criando diversos conjuntos de solução. Em cada um destes, os elementos deverão estar no mesmo nível de abstração.

No exemplo do sistema de arquivos, temos claramente 3 níveis de abstração:

nível arquivo permanente - neste nível opera-se com arquivos gravados em algum meio permanente e conhecidos por um nome simbólico.

nível arquivo em uso - neste nível opera-se com arquivos como um todo. Os arquivos em uso estão associados a algum programa sendo executado e são conhecidos por uma identificação de arquivo em uso.

nível de acesso a espaços de dados - neste nível opera-se com espaços de dados (registros) de arquivos em uso.

4.6. Completeza de interface

4.6.1. Definição

A especificação essencial de cada elemento do conjunto de solução está completa, assegurando que basta conhecer esta especificação para construir uma implementação correta e integrável para o elemento.

4.6.2. Exemplo

Consideremos a operação *ler registro*. De forma simplificada, para podermos ler um registro precisamos saber:

1- qual o arquivo em uso que contém o registro a ser lido;

2- se o arquivo em uso permite que se leia;

3- a posição inicial do espaço de dados correspondente ao registro a ser lido;

4- se esta posição inicial existe;

5- a extensão do espaço de dados correspondente ao registro a ser lido;

6- o endereço para onde será lido o registro.

Precisamos saber os requisitos que a leitura deve satisfazer, por exemplo:

1- pode ou não ler registros incompletos, por exemplo ao atingir fim de arquivo.

2- lê ou não registros contendo dados defeituosos, por exemplo erro de paridade.

3- pode ou não poluir a tela com mensagens de erro de leitura.

4- pode ou não cancelar a execução ao encontrar um erro de leitura.

Finalmente, precisamos saber as hipóteses assumidas pela solução, por exemplo:

1- os dados estarão sempre corretos.

4.6.3. Relevância

A especificação precisa e completa é fundamental para que se possa desenvolver programas com qualidade garantida. É fundamental, ainda, para que se possa desenvolver programas em equipe. No entanto, o excesso de detalhe ou até de rigor, além de aumentar em muito os custos de se criar uma especificação precisa pode meramente levar a uma substituição de uma linguagem (a linguagem de programação) por outra (a linguagem de especificação) sem ganhos na confiabilidade final. Deve-se, então, procurar linguagens de especificação simples e poderosas e especificar estritamente o que for essencial.

4.6.4. Como realizar a avaliação

Verifica-se se a especificação essencial do elemento existe e está completa. Para tal, examina-se tudo o que é público no elemento sendo considerado. Examina-se se falta alguma informação que diga respeito às propriedades públicas. Corrige-se e completa-se as especificações essenciais do elemento até que a especificação essencial seja satisfatória.

4.7. Critério ortogonalidade

4.7.1. Definição

Cada elemento do conjunto de solução resolve uma parte da abstração raiz e que não é resolvida por qualquer outro elemento do conjunto de solução. Idealmente, os elementos do conjunto de solução deveriam ser *estritamente ortogonais*, no sentido que cada um possua nenhuma superposição com qualquer um dos demais elementos do conjunto.

4.7.2. Exemplo

No exemplo do sistema de arquivos, o pacote *criar arquivo* e o *pacote abrir arquivo para gravação* não são estritamente ortogonais, uma vez que ambos levam à criação de um arquivo. De modo semelhante, os pacotes *abrir arquivo para leitura* e *abrir arquivo para gravação* não são estritamente ortogonais com o pacote *verificar se arquivo existe*. Em uma solução ortogonal teríamos:

Criar arquivo - dado um nome de arquivo não conhecido, cria um arquivo novo.

Rebatizar arquivo - dados dois nomes de arquivo onde o primeiro é o nome de um arquivo conhecido e o segundo é desconhecido, troca o nome do arquivo pelo segundo nome dado.

Verificar se arquivo existe - dado um nome de arquivo, retorna TRUE se já era conhecido.

Destruir arquivo - dado o nome de um arquivo conhecido, exclui o arquivo e torna o nome desconhecido.

Verificar se arquivo está em uso - dado o nome de um arquivo conhecido, verifica se algum programa está utilizando este arquivo.

Abrir arquivo - dado o nome de um arquivo conhecido e dado o direito de acesso solicitado, coloca o arquivo em uso.

Fechar arquivo - dada a identificação de um arquivo em uso, retira o arquivo de uso.

A seguir ilustramos o uso destes pacotes com um trecho de programa esquemático para gravar um arquivo. Caso o arquivo a gravar já exista o seu nome será trocado para "BAK":

```
Se Verificar se arquivo existe( Nome arquivo, TXT)
Então
    Se Verificar se arquivo existe( Nome arquivo, BAK)
    Então
        Destruir arquivo( Nome arquivo, BAK)
    Fim se
    Rebatizar arquivo( Nome arquivo, TXT ; Nome arquivo, BAK)
Fim se
Criar arquivo( Nome arquivo, TXT ; gravável)
IdArqUso = Abrir arquivo( Nome arquivo, TXT ; gravar)
```

4.7.3. Relevância

A ortogonalidade estrita contribui para minimizar o volume de código duplicado. A duplicação de código aumenta o esforço de desenvolvimento, controle de qualidade e manutenção, além de ser, muitas vezes, responsável por surpresas desagradáveis ao evoluir programas, mesmo quando esta evolução se der ainda durante o desenvolvimento. A evolução parcial de algumas das cópias do código leva, freqüentemente, a comportamento errático difícil de diagnosticar.

4.7.4. Como realizar a avaliação

Devem ser examinadas as especificações de todos os pares de elementos, assegurando independência em sua especificação essencial.

4.8. Viabilidade

4.8.1. Definição

Cada elemento do conjunto de solução, ou já é uma solução concreta, ou, em princípio, admite uma solução.

4.8.2. Relevância

De nada adianta continuar a desenvolver, quando se sabe que o objetivo traçado é inalcançável. A viabilidade tem por objetivo examinar, passo a passo, se os objetivos e requisitos podem, em princípio, ser alcançados. É claro que uma especificação pode parecer viável quando ela estiver sendo formulada e, mais tarde, ser demonstrada inviável. No entanto, o que se deseja é evitar o desperdício de esforço que já se sabe ser inútil.

4.8.3. Como realizar a avaliação

Examina-se cada um dos elementos do conjunto de solução e procura-se determinar se algum deles possui requisitos difíceis ou mesmo impossíveis de serem atingidos. Caso algum elemento seja inviável, pelo menos a estrutura de solução precisa ser reformulada.

5. Epílogo

Examinando os critérios acima definidos, podemos observar que necessidade, suficiência e integrabilidade avaliam a correção do conjunto de solução com relação à abstração raiz. Integrabilidade, ortogonalidade, completeza da especificação e viabilidade avaliam a qualidade de engenharia do conjunto de solução.

Estes critérios formam, claramente, um conjunto *passa/não passa*, no sentido que, qualquer falha observada precisa ser eliminada antes de se prosseguir no desenvolvimento. Ou seja, a medida (métrica) resultante de uma avaliação será ou *aprovado*, ou *recusado*. No caso de recusa, a representação examinada conterá laudos indicando onde e porque ocorreu a rejeição. Os critérios aqui apresentados admitem uma *avaliação objeti-*

va. Constituem, assim, uma alternativa eficaz para a definição da avaliação do fator coesão.

É evidente que a avaliação desses critérios dá trabalho. No entanto, este trabalho contribui não somente para a garantia de qualidade, contribui, também, para um entendimento melhor do problema a ser resolvido. Tendo em vista que a maior parte dos erros encontrados em programas entregues se deve a erros de especificação e/ou de projeto, é imediato concluir que os custos de manutenção serão significativamente menores, caso se tenha um bom entendimento do problema a resolver.

Referências bibliográficas

[DeMarco78] De Marco, T.; *Structured Analysis and System Specification*; Englewood Cliffs, NJ; Prentice Hall; 1978

[Gane79] Gane, C.; Srson, T.; *Structured System Analysis: Tools and Techniques*; Englewood Cliffs, NJ; Prentice Hall; 1979

[Jackson75] Jackson, M.A.; *Principles of Program Design*; New York; Academic Press; 1975

[Maffeo92] Maffeo, B.; *Engenharia de Software e Especificação de Sistemas*; Campus, 1992

[McMenamin84] McMenamin, S.M.; Palmer, J.F.; *Essential Systems Analysis*; Yourdon Press; 1984

[Myers72] Myers, G.J.; "Composite Design"; *Datamation*; 1972

[Page-Jones80] Page-Jones, L.; *The Practical Guide to Structured System Design*; Englewood Cliffs, NJ; Prentice Hall; 1980

[Rising92] Riding, L.; Calliss, F.W.; "Problems with Determining Package Cohesion and Coupling", *Software Practice and Experience* 22(7); Wiley; New York; julho 1992; pags 553-572

[Rocha83] Rocha, A.R.C.; *Um Modelo para a Avaliação da Qualidade de Especificações*; Tese de Doutorado; Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro; 1983

[Staa87] Staa, A.v.; *Engenharia de Programas*; Rio de Janeiro; Livros Técnicos e Científicos; 1987, 2a. edição

[STAA92] *Ambiente de Desenvolvimento de Software TALISMAN Manual de Referência*; STAA Informática, 1992

[Ward85] Ward, P.T.; Mellor, S.J.; *Structured Development for Real-Time Systems*; Yourdon Press; 1985; 3 volumes

[Wirth82] Wirth, N.; *Programming in Modula-2*; Springer; 1982

[Yourdon79] Yourdon, E.; Constantine, L.L.; *Structured Design: Fundamentals of a Discipline of Computer Programs and Systems Design*; Englewood Cliffs, N.J.; Prentice- Hall; 1979