PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 30/96

PG-07 Regras e Recomendações para Estruturas de Projeto

- Versão 1.01 -

Arndt von Staa
(Editor)

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 30/96
Editor: Carlos J. P. Lucena Setembro, 1996

PG-07 Regras e Recomendações para Estruturas de Projeto *

- Versão 1.01 -

Arndt von Staa
(Editor)

* Trabalho patrocinado pelo Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

Responsável por publicações:
Rosane Teles Lins Castilho
Assessoria de Biblioteca Documentação e Informação
PUC-Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 - Rio de Janeiro, RJ
Brasil
Tel +55-21-529 9386 Fax +55-21-511 5645
E-mail: biblio@inf.puc-rio.br www: http://www.inf.puc-rio.br

PG-07 Regras e recomendações para estruturas de projeto

Versão 1.01

editor: A.v. Staa[1]
arndt@inf.puc-rio.br

Laboratório de Engenharia de Software
Departamento de Informática
Pontifícia Universidade Católica
22453-900 Rio de Janeiro,
Brasil

Setembro 1996

PUC-RioInf.MCC 30/96

Resumo

Neste documento é apresentado um conjunto de critérios de avaliação da qualidade da estrutura de projeto. Este conjunto de critérios pode ser aplicado em todas as ocasiões em que se produz representações com vários níveis de abstração. O conjunto de critérios não depende da linguagem de representação sendo utilizada. Aplica-se igualmente bem ao projetar algoritmos, ao projetar tipos abstratos de dados, ao projetar estruturas de classes, ou ao especificar sistemas utilizando fluxos de dados ou diagramas de estado e transição.

A adoção destes critérios contribui para:

- criar boas estruturas de decomposição sucessiva.
- agregar em um único local toda a informação ou código a respeito de uma determinada abstração.
- criar estruturas de fácil manutenção.

Palavras chave: acoplamento, coesão, corretude, estrutura de projeto, manutenibilidade.

Abstract

In this document a set of design structure evaluation criteria are presented. These criteria can be applied whenever design uses techniques based on successive decomposition. The criteria are not directed towards a specific representation language or design method. They apply equally well when designing algorithms, class structures, data structures, data flow diagram structures.

When adopting this standard it is expected that:

- well organized design structures will result.
- code fragments will tend to aggregate all text concerning a given aspect of a problem to be solved.
- maintenance will be simpler and less error prone.

Keywords: cohesion, coupling, correctness, design metrics, design structure, maintainability, maintainability.

---

[1] Trabalho apoiado por: CNPq, Bolsa de Pesquisador 300029/92-6, CENPES/Petrobrás, Itautec/ Philco

Histórico de evolução

PG-07 Regras e recomendações para a avaliação de estruturas
Versão 1.00

Gestor: Laboratório de Engenharia de Software
Departamento de Informática, PUC-Rio

Arquivo: Estrut01
Editado: 16 setembro, 1996
Impresso: 16 setembro, 1996

Documentos correlatos

PG-01 Convenções para a escolha de nomes de elementos em programas C++
PG-03 Regras e recomendações para a programação em C e C++
PG-05 Regras e recomendações para a inclusão de especificações no código de programas
PG-06 Regras e recomendações para fluxos de controle em algoritmos

Versão V1.00

Editores:
Arndt von Staa (PUC-Rio)

Status: Em uso

Data homologação: 01/mar/1996
Data entrada em vigor: 01/mar/1996

Data de início da próxima revisão 02/jan/1997

Descrição de evolução

Descrição da retroação

Créditos

Revisores

André Derraik	(TeCGraf PUC)	Versão 1.0
Geraldo Machado Costa	(LES PUC-Rio)	Versão 1.0
Lincoln Nobumiti Kanamori	(Itautec Philco)	Versão 1.0
Pedro Alexandre O. Giovani	(Itautec Philco)	Versão 1.0
Pedro Jorge E. Hübscher	(LES PUC-Rio)	Versão 1.0
Renan Martins Baptista	(CENPES)	Versão 1.0
Rosa Maria Ramalho Correia	(Itautec Philco)	Versão 1.0

Apoio

CENPES Petrobrás
CNPq
Itautec Philco

Marcas registradas e nomes de produtos

MS-DOS e Windows são marcas registradas da Microsoft Corp

Sumário

1. Objetivos

Neste documento é apresentado um conjunto de critérios de avaliação para o fator de qualidade estrutura de projeto. Este conjunto de critérios pode ser aplicado em todas as ocasiões em que se produz representações com vários níveis de abstração. O conjunto de critérios não depende da linguagem de representação sendo utilizada. Aplica-se igualmente bem ao projetar algoritmos, ao projetar tipos abstratos de dados, ao projetar estruturas de classes, ou ao especificar sistemas utilizando fluxos de dados ou diagramas de estado e transição.

A adoção destes critérios contribui para:

- criar boas estruturas de decomposição sucessiva.
- agregar em um único local toda a informação ou código a respeito de uma determinada abstração.
- criar estruturas de fácil manutenção.

2. Conceitos

Uma parcela significativa dos métodos de especificação, projeto e implementação de programas utiliza níveis de abstração (decomposição sucessiva) para vencer barreiras de complexidade. Nos níveis mais abstratos, observa-se o sistema ou o programa como um todo, nos níveis intermediários, observa-se a sua arquitetura, e nos níveis finais, observa-se o projeto da estrutura do código e o próprio código. Ao utilizar níveis de abstração, precisa-se definir uma solução para cada elemento abstrato encontrado em um destes níveis. Para poder ser considerada uma boa solução, esta precisa satisfazer diversas propriedades estruturais com relação ao elemento abstrato que pretende resolver. O objetivo do conjunto de critérios é definir o que se entende por uma boa solução. Como a propriedade *boa solução* pode ser interpretada de várias maneiras – é ambígua –, procuramos aqui caracterizar como boa solução, uma que facilite a manutenção e o entendimento da solução.

2.1 Especificação e controle de qualidade

Para termos certeza que vamos construir um software com uma determinada qualidade, precisamos saber de antemão quais são as propriedades que determinam esta qualidade. Precisamos saber definir e medir, ou avaliar, propriedades determinantes da qualidade. Estas propriedades dividem-se em dois grandes grupos:

i. *qualidade de serviço*, descreve as propriedades que o software, ou componente, deve possuir para que o serviço por ele produzido seja aceitável. Determinam a qualidade do serviço de um componente as propriedades percebidas pelo usuário deste componente. Entendemos aqui o termo *usuário de um componente* de forma ampla, abrangendo pessoas utilizando o aplicativo que contém este componente, bem como outros componentes interagindo com o componente em questão. A qualidade de serviço é definida através da especificação essencial[2]. Esta define precisamente o que o software ou componente deve realizar, sem se preocupar com a forma da realização.

ii. *qualidade de engenharia*, descreve as propriedades que a implementação do software ou componente deve ter para ser uma solução aceitável. Determinam a qualidade de engenharia as propriedades do software ou de seus componentes percebidas pelos construtores e mantenedores deste software ou destes componentes. A qualidade de engenharia é alcançada através de especificações de arquitetura e de projeto. Conseqüentemente, está diretamente relacionada com a organização e composição da forma com que a especificação essencial é implementada – *reificada*[3].

Ainda a um nível muito abrangente estas duas classes de qualidade podem ser decompostas nos objetivos de qualidade que identificamos a seguir[4]. A *qualidade de serviço* é definida em termos de:

confiabilidade o sistema (componente) produz resultados confiáveis (tipicamente: corretos, dentro das margens de tolerância) sempre que solicitado.

segurança os riscos (tipicamente: danos pessoais, ecológicos, materiais, ou financeiros) decorrentes do uso do sistema (componente) são aceitáveis.

utilidade o sistema (componente) resolve eficazmente os problemas do usuário, ajudando-o a realizar as suas tarefas mais rapidamente, de forma mais abrangente ou com melhor qualidade.

utilizabilidade o sistema (componente) pode ser utilizado com segurança por pessoas com o nível de formação estipulado na especificação do sistema (componente).

rentabilidade de serviço o sistema (componente) requer recursos computacionais compatíveis com a complexidade dos problemas sendo resolvidos e com o valor dos serviços prestados pelo sistema (componente).

---

2 O termo *especificação essencial* é 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 não a apliquemos somente aos elementos da especificação de um sistema.

3 *Reificar*: tornar uma abstração ou especificação real (existente), ou seja, criar um sistema ou componente em conformidade com uma determinada especificação..

4 Seguimos de perto a estrutura do modelo de qualidade descrito em [Rocha 87]. No entanto, os itens que constituem o modelo são diferentes.

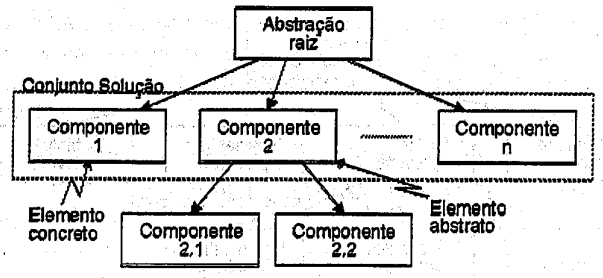

Figura 1. Ilustração dos conceitos de estrutura

A *qualidade de engenharia* é definida em termos de:

evolutibilidade o sistema (componente) pode ser corrigido, aperfeiçoado, adaptado e evoluído sem perda de qualidade de serviço ou de engenharia.

mensurabilidade as propriedades relevantes do sistema (componente) podem ser medidas.

rentabilidade de engenharia o custo de desenvolvimento é compatível com a complexidade do sistema (componente).

rentabilidade de manutenção as alterações requeridas podem ser realizadas rápida e economicamente, assegurando a manutenção ou a melhoria do nível de qualidade do sistema (componente).

Na realidade estas propriedades devem ser satisfeitas a cada nível de abstração. Por exemplo para que uma *função* produza um serviço de boa qualidade, ela precisa ser confiável, precisa resolver um problema relevante, precisa possuir uma interface adequada às funções cliente, precisa ser econômica no uso de recursos computacionais e humanos. Do ponto de vista de qualidade de engenharia, a função deve poder se mantida sem que isto implique alterações em outras funções, seu desempenho deve poder ser medido, deve ser pequeno o esforço necessário para o entendimento de seu código e/ou da alteração deste. De forma semelhante, para que uma *classe* produza um serviço de boa qualidade, ela precisa ser confiável, precisa reificar integralmente um único conceito do mundo real, precisa ter uma interface consistente com as classes de que herda, que herdam dela, e com quem interage. Finalmente, precisa ser econômica no consumo de recursos computacionais. Estes dois exemplos ilustram como as propriedades acima identificadas podem ser ajustadas a cada nível de abstração.

Enquanto estivermos produzindo ou mantendo especificações, arquiteturas e projetos, utilizamos documentos – *representações*. Estas, em geral, não permitem a observação direta do comportamento do componente sendo especificado ou projetado. Mesmo assim temos que controlar a qualidade das representações. Isto é realizado através da avaliação das representações segundo determinados *critérios de qualidade*. Estes estabelecem mais do que meros padrões de avaliação da qualidade. A simples existência da definição e a adoção desses critérios conduz os desenvolvedores a produzir representações e, ao final, implementações, possuindo a qualidade estabelecida pelo critério.

2.2 Componentes de uma solução baseada em refinamentos sucessivos

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 classes, estruturas de chamadas de funções (estruturas modulares [Yourdon79], estruturas de pacotes[5]), e de estruturas de blocos (algoritmos). No caso de uma estrutura de blocos, uma pseudo-instrução é descrita em termos de outras pseudo-instruções e/ou de operações concretas. Ou seja, cada abstração é realizada, ou implementada, através de um conjunto de elementos que, por sua vez, podem ser abstrações em um nível de abstração menor ou, então, ser implementações acabadas (componentes concretos).

[5] Um *pacote* corresponde a uma função, método, subrotina, procedimento, sub-programa, macro. Utilizamos um novo termo para nos tornar independente da terminologia utilizada em linguagens de programação específicas. Cabe salientar o termo *package* utilizado em Ada não corresponde ao conceito pacote utilizado neste texto.

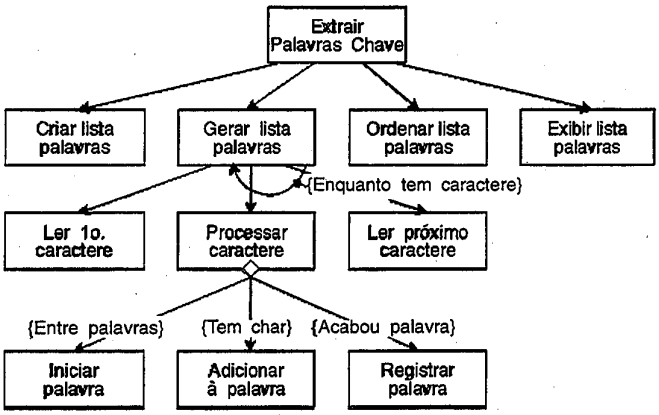

Figura 2. Ilustração de uma estrutura de algoritmo

De modo semelhante, ao projetarmos estruturas de dados complexas, lançamos mão de módulos (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*, entre outras, e dos tipos de dados *Identificador do símbolo*, *Nome de símbolo*, e *Tabela de símbolos*.

Concluindo, quando especificamos, projetamos ou implementamos sistemas de programação, trabalhamos com *abstrações raiz* e com *conjuntos de solução* que resolvem ou implementam estas abstrações, ver Figura 1. Os elementos de um conjunto de solução podem, por sua vez, ser abstrações raiz de outros conjuntos de solução, gerando assim uma *estrutura de solução*, ver Figura 2.

Uma abstração raiz é um elemento qualquer complexo demais para que se possa dar uma solução direta. Exemplos de abstrações raiz:

um processo definindo uma funcionalidade complexa a ser a descrita por um diagrama de fluxo de dados de nível mais baixo.

uma classe implementando uma estrutura de dados manipulada por um conjunto de métodos.

um bloco de código implementando uma pseudo-instrução definida em termos de outros blocos e/ou de código fonte.

uma tarefa a realizada em termos de uma rede de precedência formada por tarefas mais elementares.

Deve estar claro que a abstração raiz não precisa ser da mesma natureza que os elementos do conjunto de solução. Estes, por sua vez, podem pertencer a uma variedade de categorias. Por exemplo, no caso de uma classe, a interface da classe – o conjunto de solução da classe – pode conter constantes, dados, tipos e métodos.

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*. todos operando sobre o depósito de dados *Contas correntes*. 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 um algoritmo realizando aproximações sucessivas, podemos imaginar um conjunto de solução contendo os seguintes elementos: *Validar dados de entrada*, *Calcular a primeira aproximação*, *Calcular próxima aproximação*, *Verificar se atingiu precisão suficiente*, *Verificar se esgotou o número de iterações permitidas*.

Os elementos de diferentes conjuntos 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 ao resolver diferentes abstrações raiz. É permitido, ainda, que um elemento de um conjunto de solução seja um antecessor deste conjunto, formando uma estrutura recursiva, ver Figura 3.

Quando um conjunto de solução estiver completamente definido, a correspondente abstração raiz **está implementada**. Ou seja, mesmo que os elementos do conjunto de solução sejam abstratos, o problema correspondente à abstração raiz estará resolvido.

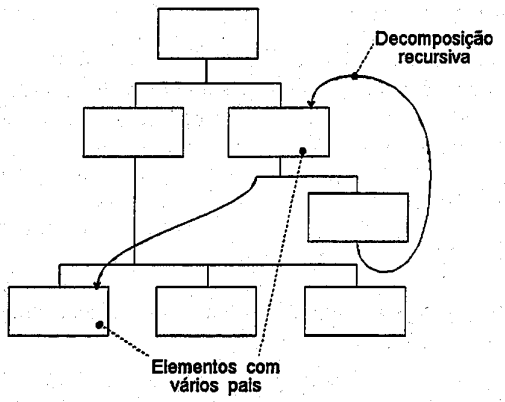

Figura 3. Ilustração de uma estrutura genérica

Cada conjunto de solução surge como conseqüência de um passo de projeto. 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 a estrutura de solução resultante satisfazer propriedades de qualidade estrutural.

A conseqüência imediata dos conceitos introduzidos nesta seção, é a necessidade do conjunto de solução corresponder a uma solução exata da abstração raiz. Entretanto, a solução definida pelo conjunto de solução não basta ser correta e completa, é necessário também que seja uma *boa solução*.

2.3 Critérios de avaliação da estrutura

Seguindo o modelo de avaliação da qualidade descrito em [Rocha 87], definimos *estrutura* como um *fator de qualidade* composto pelos seguintes critérios de avaliação:

Completeza de interface a especificação essencial de cada elemento do conjunto de solução esta completa. Basta conhecer esta especificação para poder desenvolver uma implementação correta do elemento. Na realidade este é um sub-fator avaliado através de diversos critérios, que serão descritos mais adiante.

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

Suficiência o conjunto solução como um todo resolve completamente a abstração raiz.

Integrabilidade é possível combinar os elementos do conjunto de solução sem necessitar de conversores de interface.

Minimalidade não existe subconjunto 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 não possuam superposição.

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

Examinando os critérios acima definidos, podemos observar que *completeza de interface*, *necessidade*, *suficiência* e *integrabilidade* avaliam a correção do conjunto de solução com relação à abstração raiz. *Minimalidade* e *ortogonalidade* avaliam a organização 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. Ao terminar o desenvolvimento da representação, a estrutura completa deve ser revista. Caso não seja satisfatória, deve-se corrigi-la. Desta forma, deficiências estruturais da representação serão eliminadas. Com elas são eliminadas diversas fontes de erros.

Viabilidade antecipa problemas que poderiam surgir durante o processo de desenvolvimento, tais como: é impossível resolver o problema formulado, o tempo de resposta da solução adotada é excessivamente alto, ou a interface do usuário é excessivamente complexa. Uma vez desenvolvido, a avaliação do desempenho através de medições, de simulações, ou de modelos matemáticos, é utilizada para avaliar a capacidade, para sintonizar o software e para determinar gargalos de desempenho.

2.4 Avaliação da interface

Além de avaliar a estrutura propriamente dita, deve-se avaliar as especificações das interfaces de cada um dos elementos do conjunto de solução.

A interface de um elemento de um dado conjunto de solução é composta por vários *itens*. Por exemplo, parâmetros de função, tipos de dados, variáveis globais, e estados da estação de trabalho. Este *conjunto de itens* deve ser examinado utilizando alguns dos critérios já apresentados para a avaliação da estrutura e outros critérios específicos de interface.

A interface de cada elemento do conjunto de solução deve ser avaliada utilizando os critérios:

Necessidade de item o item de interface é necessário para que se possa desenvolver o elemento do conjunto de solução.

Suficiência de itens o conjunto de todos os itens de interface do elemento permitem desenvolver o correspondente elemento do conjunto de solução.

Minimalidade de itens não é possível agregar subconjuntos de itens de modo a formar um único item composto possuindo boa estrutura.

Ortogonalidade dos itens cada item da interface se refere a uma propriedade da interface que nenhum outro item considera.

Encapsulamento a interface não externa propriedades de implementação do elemento.

Flexibilidade a interface permite a generalização do uso do elemento, mantendo única a intenção ou objetivo do elemento.

O critério de *suficiência de itens* equivale, em linhas gerais, ao critério completeza da interface definido na seção anterior. *Necessidade de item* e *suficiência de itens* precisam ser satisfeitos para que o elemento possa ser corretamente implementado. *Minimalidade de itens* e *ortogonalidade dos itens* procuram restringir o tamanho da interface. Quanto menor for esse tamanho, mais fácil será aprender como utilizar o elemento. *Encapsulamento* e *flexibilidade* procuram aumentar a chance de reúso do elemento em outros conjuntos de solução, além de tornar o elemento menos sujeito a efeitos colaterais de alteração[6].

2.5 Processo de desenvolvimento com controle de qualidade simultâneo

A avaliação da estrutura e das interfaces 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 tiver sido completamente gerada ou alterada, esta estrutura deverá ser avaliada. Reduzem-se, assim, as ocasiões em que são desen-

---

[6] Ocorre um *efeito colateral de alteração* em um elemento *A* se, ao alterar *A*, o comportamento de um outro elemento *B* for afetado, sem que se tenha realizado qualquer alteração explícita em *B*.

volvidos elementos a partir de especificações incorretas ou incompletas. De nada adianta continuar detalhando ou desenvolvendo, quando já se sabe que a solução proposta é inadequada.

É evidente que a avaliação da qualidade consome esforço. No entanto, este esforço contribui para a garantia de qualidade, contribui para a diminuição da probabilidade da ocorrência de erros durante o uso, e contribui, também, para um melhor entendimento do problema a ser resolvido. Tendo em vista que a maior parte dos defeitos encontrados em programas entregues se deve a erros de especificação ou de projeto, pode-se concluir que a adoção destes critérios contribui significativamente para reduzir os custos de desenvolvimento (menos defeitos a remover durante o desenvolvimento), para reduzir os custos de manutenção (menos defeitos remanescentes nos elementos entregues), e para reduzir custos decorrentes de erros ocorridos durante a execução (menos danos provocados por defeitos exercitados durante a execução).

Sistemas de programação modernos tendem a ser avessos a testes aprofundados, ou por conterem componentes de execução concorrente, ou por serem dependentes de eventos externos cuja ordem não é previsível, ou pela simples dimensão dos sistemas, ou por combinações desses fatores. Por outro lado, os custos decorrentes de erros de um sistema podem ser imensos. Por exemplo, quanto custa para uma empresa um sistema *on-line* parado? Quanto custa para uma fábrica de máquinas de lavar a substituição dos controladores programados quando estes apresentam defeitos de programação? Os riscos decorrentes de erros de sistemas também podem ser enormes. Por exemplo, você gostaria de estar voando em uma aeronave cujo software de controle é defeituoso? Considerando os riscos e custos induzidos pelo uso de software imperfeito e a dificuldade de realizar testes, torna necessário desenvolver sistemas que estejam *corretos por construção*. Para alcançar este ideal é necessário que as especificações, o projeto e a implementação possuam elevado grau de qualidade. Isto somente pode ser conseguido utilizando processos de desenvolvimento nos quais a qualidade esperada é claramente estabelecida antes de iniciar uma tarefa de desenvolvimento, e nos quais os resultados cada tarefa, por menor que seja, passe por uma verificação segundo critérios definidos [Humphrey 91][Humphrey 95][Paulk 93][Rocha 87].

Cabe salientar que a revisão da especificação de uma abstração raiz, implica a 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 aproximações sucessivas convergindo para uma solução aceitável*. Neste, à medida que se vai adicionando conjuntos de solução e/ou revendo especificações, partes da estrutura de solução tornam-se estáveis, enquanto que outras partes da estrutura de solução ainda continuam sujeitas a alterações. Eventualmente este processo de desenvolver, avaliar, rever, desenvolver, etc. convergirá para uma boa solução. A convergência se dá tão mais rapidamente quanto maior o rigor da avaliação a cada passo. Note que, embora pareça, isto não quer dizer que estamos propondo um método de trabalho do gênero *tentativa e erro*. O processo proposto admite, realisticamente, que erros poderão ser cometidos ao criar, examinar ou manter e, portanto, temos que estar preparados para encontrar os correspondentes defeitos e eliminá-los de uma forma sistemática. Quanto menor o tempo decorrido desde a introdução de um defeito até a observação e eliminação deste, significativamente menor será o esforço de desenvolvimento. Significativamente menor será, também, o volume de defeitos remanescentes em um software dado como terminado.

2.6 Descrição do exemplo

Para ilustrar os critérios de avaliação do fator estrutura, utilizaremos 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 conjunto de solução a seguir. A especificação está propositalmente vaga, incompleta e ambígua para poder ilustrar como os critérios auxiliam na geração de uma especificação precisa.

Descritor de arquivo aberto

- identifica o nome do arquivo em uso
- determina a localização física do arquivo no meio de armazenamento
- define os direitos de acesso ao arquivo para o programa usuário

Abrir arquivo para leitura

- procura o nome de um arquivo no catálogo
- preenche o descritor de arquivo aberto com os dados de acesso físico do arquivo
- posiciona no início do arquivo
- preenche o descritor de arquivo aberto com os direitos de acesso do usuário do programa, incluindo o acesso ler somente.

Abrir arquivo para gravação

- procura o nome de um arquivo no catálogo
- destroi o arquivo gravado caso exista
- introduz no catálogo de arquivos conhecidos o nome do arquivo a gravar
- determina o endereço físico de origem do arquivo
- preenche o descritor de arquivo aberto com os dados de acesso físico do arquivo
- posiciona no ponto corrente inicial a partir do qual se dará a gravação
- preenche o descritor de arquivo aberto com os direitos de acesso do usuário do programa, incluindo o acesso gravar somente.

Ler registro

- lê *n* bytes a partir da posição corrente definida no descritor de arquivos
- avança a posição corrente de *n* bytes

Gravar registro

- grava *n* bytes a partir da posição corrente definida no descritor de arquivos
- avança a posição corrente de *n* bytes

Imprimir registro

- imprime *n* bytes a partir da posição corrente definida no descritor de arquivos
- avança a posição corrente de *n* bytes

Posicionar registro

- define a posição corrente

3. Definição da norma

3.1 Regras do processo de desenvolvimento

Regra 1: Verifique a satisfação de todos os critérios de qualidade descritos nesta norma sempre que tiver concluído a criação ou alteração de um conjunto de solução.

Regra 2: Caso seja necessária a alteração de uma abstração raiz, proceda da seguinte maneira:

1. sendo permitido alterar a abstração raiz, altere-a e reavalie os conjuntos de solução nos quais esteja incluída.
2. não sendo permitido alterar a abstração raiz, produza uma solicitação de alteração, detalhando e justificando a adição à abstração raiz.

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

- modificar a composição do conjunto de solução.
- rever a especificação de um ou mais elementos do conjunto de solução.
- rever a especificação da abstração raiz. Isto corresponde, em última análise, a modificar o conjunto de solução ao qual pertence a abstração raiz.
- uma combinação destas.

Muita vezes, ao criar um conjunto de solução observa-se que a correspondente abstração raiz está mal ou incompletamente definida. Podem ocorrer as seguintes situações:

i. quando a abstração raiz é uma *especificação aceita*[7] em etapa anterior, não se pode simplesmente modificá-la, uma vez que isto criaria problemas de consistência com outras tarefas do projeto. Neste caso deve ser gerada uma solicitação de alteração descrevendo e justificando a alteração pedida. Cabe agora à gerência do projeto determinar quando e se esta solicitação será efetuada.

ii. quando a abstração raiz de acesso irrestrito é a *raiz de toda a estrutura*, pode-se alterá-la. Neste caso reavalia-se somente a nova especificação de interface.

iii. quando a abstração raiz de acesso irrestrito é membro de um ou mais outros conjuntos de solução, pode-se efetuar a alteração. Porém, a revisão da especificação de uma abstração raiz, implica a revisão de todos os conjuntos de solução de que esta abstração raiz faça parte, além do conjunto de solução da própria abstração raiz.

Eventualmente este processo de desenvolver, avaliar, rever, desenvolver, etc. convergirá para uma boa solução. A convergência se dá tão mais rapidamente quanto maior o rigor da avaliação a *cada* passo.

3.2 Regras do fator completeza de interface

Regra 3: Assegure que cada elemento do conjunto de solução possua uma especificação.

Exceção 4: Quando o nome do elemento for suficiente para identificar a intenção[8] (objetivo) do elemento, não é necessária a especificação.

Regra 5: Assegure que a intenção de cada elemento do conjunto de solução esteja claramente definida.

Regra 6: Assegure que cada elemento corresponda a exatamente uma intenção. Caso não corresponda, particione o elemento em tantos novos elementos quantas forem as intenções identificadas.

Regra 7: Assegure que as especificações de cada um dos elementos satisfaçam os critérios:

1. Definição;

---

[7] Uma *especificação aceita* deve estar incluída em um sistema de controle de versões, e deve estar protegida contra alterações. Para alterar, a gerência precisa autorizar a alteração e criar uma nova versão ou modificação.

[8] *Intenção*: o conjunto dos motivos do autor ao escrever uma obra, em oposição à obra realizada [Aurélio Eletrônico]. No caso corresponde aos motivos pelo quais o elemento foi incluído no conjunto de solução.

2. Necessidade de item;
3. Suficiência de itens;
4. Estrutura de itens;
5. Ortogonalidade dos itens;
6. Encapsulamento;
7. Flexibilidade.

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, além de aumentar em muito os custos de se criar e manter uma especificação precisa, pode meramente aumentar os custos sem ganhos na confiabilidade final. Deve-se procurar redigir especificações simples e ater-se a especificar estritamente o que for essencial. A norma *PG-05 Regras e recomendações para inclusão de especificações no código* define a forma e o conteúdo das especificações (os elementos e a organização da especificação). A presente norma define as propriedades do conteúdo de especificações.

A especificação de cada elemento do conjunto de solução deve estar completa, assegurando que basta conhecer esta especificação para construir uma implementação correta e integrável para o elemento. Em alguns casos o elemento é de tal forma simples que a especificação acaba sendo redundante com o nome do elemento. Isto é particularmente comum no caso de pseudo-instruções identificando blocos de código fonte executável.

Cada elemento deve corresponder a exatamente uma intenção, ou objetivo. Esta intenção é identificada no nome dado ao elemento e na definição, ou descrição, do elemento. Caso o elemento possua um nome complexo, ou o nome contenha conectores tais como *e*, *ou*, *senão*, *mas*, *também*, relativas a propriedades marcantes do elemento, é provável que o elemento implemente mais do que uma intenção. Por exemplo, uma função cuja funcionalidade é: *Ler e imprimir registro* deve ser particionada em duas: *Ler registro* e *Imprimir registro*. Note que abstrações bem definidas, usualmente possuem nomes definidos por uma frase simples – menos de 16 palavras –, onde estes nomes capturam integralmente a intenção do elemento.

Exemplo

Considere a operação *Ler registro*. Para podermos redigir esta função precisamos conhecer os dados manipulados, por exemplo:

- como informar qual o arquivo em uso que contém o registro a ser lido.
- como informar qual a posição inicial no arquivo em uso a partir da qual se vai ler.
- como informar a posição inicial do espaço de dados onde será armazenado o registro a ser lido.
- como informar a extensão do espaço de dados correspondente ao registro a ser lido.
- como saber quanto foi lido.
- quais são e o significado de cada possível condição de retorno.
- como saber qual a condição de retorno da leitura, e quais as condições que podem ocorrer.

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

- o que fazer se o arquivo em uso, ou dispositivo a ele associado, não permite que se leia.
- se pode ou não ler registros incompletos, por exemplo ao atingir fim de arquivo.
- se pode ou não ler registros contendo dados defeituosos, por exemplo erro de paridade.
- se o cursor de leitura pode ou não estar posicionado além do tamanho do arquivo.
- se pode ou não poluir a tela com mensagens de erro de leitura; por exemplo: "dispositivo sem disco".
- se pode ou não cancelar a execução ao encontrar um erro de leitura.

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

- se é assumido que os dados estarão sempre corretos.

3.3 Regras do critério necessidade

Regra 8: Verifique para cada um dos elementos contidos no conjunto de solução:
1. qual a parcela da abstração raiz que é resolvida pelo elemento.

2. se existe alguma parcela do elemento sendo examinado que não esteja diretamente comprometida com a solução da abstração raiz.

Regra 9: Todos os elementos que não resolvam alguma parcela da abstração raiz devem ser retirados do conjunto de solução.

Regra 10: Todos os elementos que resolvam integralmente a abstração raiz devem ou ser modificados para que passem a resolver uma parcela, ou ser retirados do conjunto de solução.

Regra 11: Todos os elementos que possuam parcelas não comprometidas com a abstração raiz devem ser particionados.

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. Em diversas ocasiões elementos desnecessários em um conjunto de solução estão mal posicionados. Ou seja, movendo estes elementos para outros conjuntos de solução, obtém-se uma boa estrutura.

Elementos que resolvem integralmente o conjunto de solução são defeitos de estrutura, uma vez que não particionam o problema em problemas menores. Note que ao utilizar algoritmos recursivos, a chamada recursiva pode parecer, a primeira vista, um caso destes. Na realidade não o será, desde que a chamada se aplique a um subproblema. Por exemplo, ao caminhar em uma árvore, a função de caminhamento é ativada para todas as subárvores de um nó. Neste caso ela está explicitamente resolvendo um subproblema. No entanto, ao caminhar em um grafo cíclico sem marcar os nós já examinados, não se está resolvendo um subproblema, uma vez que se pode retornar a um nó já examinado e, como não existe indicação "já visitado", continuar a percorrer o grafo a partir desse nó.

Elementos do conjunto de solução podem possuir uma funcionalidade mais abrangente do que a estritamente necessária. Neste caso devem ser particionados. Note que isto não conflita com flexibilidade. Um elemento flexível implementa uma única funcionalidade porém capaz de ser adotada em uma gama de situações. Por exemplo, uma função *f(d, v)* que processa um vetor *v* de tamanho *d* é uma função flexível. Se ela for utilizada em um conjunto onde a dimensão *d* é constante, não faz sentido querer modificar *f* meramente para que se deixe de passar o parâmetro *d*.

Exemplo

No sistema de arquivo estão definidas as funções *Imprimir registro* e *Gravar 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 deve ser a mesma operação do ponto de vista externo. Isso torna desnecessária a função *Imprimir registro*. Como conseqüência desta eliminação pode ocorrer que uma função não possa ser realizada por estar associada a um dispositivo que impeça a execução. Ou seja, é necessário que as funções possam informar anomalias de execução que porventura tenham ocorrido. Resumindo, as funções *Ler registro*, *Gravar registro* e *Posicionar registro* devem receber pelo menos os seguintes itens de especificação a mais:

- a função é independente de dispositivo.
- a função retorna um valor do tipo *ARQ_tpCondRet* indicando como foi concluída a sua execução.

3.4 Regras do critério suficiência

Regra 12: Verifique se existem parcelas da abstração raiz que não estejam sendo resolvidas por algum dos elementos do conjunto de solução. Caso existam, inclua elementos no conjunto de solução para resolver estas parcelas.

Regra 13: Verifique se é possível encontrar um elemento que se possa adicionar ao conjunto solução de modo que a intenção da abstração raiz se torne mais completa. Caso exista, proceda como descrito na Regra 2.

Recom. 14: Procure manter os pares de operações parentetisantes – definidas mais adiante – em um mesmo conjunto de solução.

Recom. 15: Procure manter todos os elementos de uma função geradora em um conjunto de solução.

Especificações e projetos incompletos são uma fonte de defeitos cuja eliminação é cara e desgastante. Soluções incompletas levam muitas vezes a escolhas de estruturas de dados ou de algoritmos inapropriados. A correção de tais falhas pode ter custos proibitivos.

Os elementos do conjunto solução devem resolver completamente a abstração raiz. Ou seja, não pode existir parcela da abstração raiz que não seja resolvida por algum dos elementos do conjunto de solução.

Muita vezes, ao criar um conjunto de solução observa-se que a abstração raiz poderia ser generalizada. Neste caso deve-se observar o que foi descrito na seção *Processo de desenvolvimento com controle de qualidade simultâneo*.

Operações parentetisantes são pares de operações onde uma desfaz o efeito da outra. São exemplos:

- abrir e fechar arquivos;
- criar e destruir arquivos.
- alocar e desalocar espaços de dados;
- inserir e excluir elementos em uma estrutura de dados ou em um arquivo de acesso direto.
- construir e destruir classes;

Se a primeira vista uma das operações parentetisantes não pareça ser necessária, inclua o nome da função como um comentário e justifique a não implementação. São raros os casos nos quais o par não é necessário.

Funções geradoras[9], ou iteradores, são coletâneas de funções e tipos de dados que coordenam o processamento iterativo através de estruturas de dados. Procure manter os itens destas funções juntos em um somente conjunto de solução.

Exemplo

O módulo sistema de arquivos utiliza os conceitos: *arquivo* (um espaço de dados contido em algum meio de armazenamento, por exemplo disco), *arquivo em uso* (um descritor informando que programas estão utilizando o arquivo, e propriedades instantâneas deste uso) e *registro* (um espaço de dados contido no arquivo e respectiva descrição de acesso). Estes três conceitos, correspondem a tipos e são necessários, embora não estejam todos na definição original. Também não se encontra na definição original a função *Destruir arquivo*, apesar de arquivos poderem ser criados ao serem abertos para a gravação.

3.5 Regras do critério integrabilidade

Regra 16: Verifique se são consistentes as especificações de interface de todos os elementos do conjunto de solução que se inter-relacionem. Corrija-as caso não o sejam.

Regra 17: Verifique se as interfaces dos elementos do conjunto de solução são consistentes com as interfaces da abstração raiz. Corrija-as caso não o sejam, obedecendo a Regra 2 caso a correção proposta possa afetar a abstração raiz.

A abstração raiz deve corresponder a uma intenção única, completa e bem definida. Conseqüentemente, existirão interfaces entre praticamente todos os possíveis pares de componentes deste conjunto. Caso o conjunto de solução não seja fortemente inter-relacionado, ou ele não satisfaz as regras de minimalidade descritas mais adiante, ou estão faltando itens de especificação nos elementos do conjunto de solução.

As interfaces entre os elementos devem estar explicitamente especificadas. Para assegurar que seja possível compor uma implementação correta da abstração raiz a partir das implementações de cada um dos elementos do conjunto de solução é necessário e suficiente:

i. que as interfaces de cada par de elementos que se inter-relacione sejam consistentes entre si.
ii. que as interfaces dos elementos que se inter-relacionam via a interface da abstração raiz sejam consistentes com esta interface.
iii. que cada elemento implemente exatamente a sua especificação de interface.

---

[9] A norma *PG-06 Regras e recomendações para fluxos de controle em algoritmos* contém a definição e explicações relativas a funções geradoras.

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

Exemplo

No módulo sistema de arquivos temos que definir uma sintaxe para o nome simbólico de arquivo. Temos, que escolher uma estrutura de dados – *diretório* – para registrar os nomes e espaços dos arquivos conhecidos. Temos que escolher uma estrutura de dados – *descritor de arquivo em uso* – para controlar os arquivos que estão em uso. Finalmente, temos que escolher uma estrutura de dados para controlar 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*, *operação não vale para dispositivo*, etc.

As estruturas de dados *diretório*, *arquivo em uso* e *acesso a registro* são estruturas internas e, portanto, devem ser encapsuladas. Embora encapsuladas, precisa ser de conhecimento público que existem, uma vez que estas estruturas determinam os estados utilizados na especificação formal dos métodos públicos. O que é encapsulado é somente a organização física destas estruturas. Já o nome de arquivo, a identificação de arquivo em uso e as possíveis condições de retorno são tipos públicos e devem ser exportados. Note que ao programar orientado a objetos, a identificação do arquivo em uso é o próprio objeto. A classe a parir da qual é instanciado o objeto é, em última análise, um tipo público.

A integração das diversas operações se dá através destas estruturas de dados, ou tipos. 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.

3.6 Regras do critério minimalidade

Regra 18: Procure subconjuntos formados por dois ou mais elementos do conjunto de solução e que correspondam a uma intenção completa. Caso tenha encontrado um subconjunto destes:

1. crie um nome para este subconjunto, refletindo precisamente a sua intenção, este elemento passa a ser uma nova abstração raiz.
2. elimine o subconjunto do conjunto de solução original.
3. coloque no lugar do subconjunto eliminado a nova abstração raiz.
4. o subconjunto eliminado passa a ser o conjunto de solução da nova abstração raiz.
5. repita até que não encontre mais subconjuntos que tenham identidade própria.

Recom. 19: Assegure que todos os elementos de um mesmo conjunto de solução estejam no mesmo nível de abstração.

Recom. 20: Assegure que todos os elementos de um mesmo conjunto de solução sejam fortemente inter-relacionados.

Quanto menor a cardinalidade dos conjuntos de solução mais fácil será o entendimento, a avaliação da correção e a manutenção destes conjuntos. Além disso, a possibilidade de particionar um conjunto de solução em vários outros, é um indicador de que é pequena a interdependência (coesão) entre os elementos do subconjunto fatorado e do restante do conjunto de solução. O particionamento é quase sempre possível caso os elementos conjunto de solução não sejam fortemente inter-relacionados[10].

Procura-se subconjuntos formados por dois ou mais elementos e que possam representar um conceito próprio. Usualmente isto acontece quando alguns dos elementos possuem especificações com interseções grandes, por

[10] De uma forma intuitiva, um *conjunto é fortemente inter-relacionado*, caso os elementos do conjunto se relacionem com outros elementos do conjunto de tal maneira que:

1. se possa caminhar de um elemento do conjunto para qualquer outro, utilizando estas relações;
2. mesmo eliminando alguns poucos relacionamentos, continua-se podendo atingir, através de caminhamento, todos os elementos a partir de qualquer um deles.

exemplo tratam de um mesmo assunto, ou quando possuem interfaces com definições semelhantes. Sendo possível encontrar um subconjunto com identidade própria, substitui-se esse subconjunto por uma nova abstração raiz que reflita essa identidade. O subconjunto encontrado passa a ser o conjunto solução desta nova raiz. A substituição do subconjunto pela nova abstração raiz, reduz a cardinalidade do conjunto de solução original, aumentando a hierarquia da solução final dada ao problema. Caso não se encontre um subconjunto com identidade própria, o conjunto é minimal[11], uma vez que não é possível reduzi-lo através da substituição de um subconjunto por um único elemento.

Ao invés de laboriosamente procurar subconjuntos com identidade própria, 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.

Exemplo

No exemplo do sistema de arquivos é fácil observar que a interdependência entre as funções *Ler registro* e *Destruir arquivo* é muito pequena. Já entre *Posicionar registro* e *Ler registro* é muito grande. Examinando mais cuidadosamente podemos observar 3 níveis de abstração:

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

ii. nível *arquivo em uso* – neste nível ainda se opera com arquivos como um todo. Os arquivos em uso estão associados a algum programa em execução e são conhecidos por uma identificação associada a um descritor de arquivo em uso.

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

Estes 3 níveis podem ser agregados em módulos, ou classes:

Gerenciar arquivos, composto por
- Tipos de dados
  - *Nome de arquivo*
  - *Condição de retorno arquivo*
  - *Estrutura de armazenamento de arquivos, encapsulado*
- Funções
  - *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*
  - *Condição de retorno arquivo em uso*
- Funções
  - *Verificar se arquivo está em uso*
  - *Abrir arquivo para leitura*
  - *Abrir arquivo para gravação*
  - *Abrir arquivo para alteração (ler e gravar)*
  - *Fechar arquivo*

Acessar registros de arquivo, composto por
- Tipos de dados
  - *Descritor de acesso a registro*
  - *Condição de retorno acesso a registro*
- Funções

---

[11] Um conjunto é *minimal* segundo alguma propriedade do conjunto como um todo se, ao retirar um único elemento, o conjunto deixa de satisfazer esta propriedade. No caso geral podem existir diversos conjuntos minimais satisfazendo uma mesma propriedade. Desses, os de menor cardinalidade serão conjuntos *mínimos*. De maneira geral o esforço de procurar conjuntos mínimos não é compensado pelas vantagens advindas do fato de ser mínimo. Por esta razão procuramos encontrar meramente algum conjunto minimal.

Verificar se registro existe
Posicionar registro
Ler registro
Gravar registro

No exemplo acima convém salientar o particionamento de condição de retorno em 3 diferentes grupos de condições. Isto se deve à recomendação de manter juntos somente os elementos que são fortemente interdependentes.

O exemplo a seguir ilustra a mecânica definida na Regra 18. Suponha a seguinte estrutura:

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 seguinte 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

3.7 Regras do critério ortogonalidade

Regra 21: Assegure que cada elemento do conjunto de solução resolve uma parte da abstração raiz que nenhum outro elemento resolve.

Recom. 22: Procure organizar o conjunto de solução de tal modo que cada elemento somente resolva itens da especificação da abstração raiz que nenhum outro elemento resolve.

Regra 23: Ao projetar ou implementar programas, sendo necessária a redundância de componentes de programas, justifique a necessidade da redundância e incorpore esta justificativa em um comentário. Este comentário deve ficar na abstração raiz correspondente.

Cada elemento do conjunto de solução deve resolver uma parte da abstração raiz que não é resolvida por qualquer outro elemento do conjunto de solução. Caso haja superposições, fatalmente existirão duplicações de formas de resolver. Isto por sua vez obriga a efetuar alterações semelhantes em diversos lugares sempre que se fizer uma manutenção em um dos elementos que possui superposição. Este tipo de problema ocorre com grande freqüência sempre que se utiliza técnicas de programação baseadas em *copiar e colar*. Embora esta forma de reúso diminua o volume de trabalho ao criar um elemento, aumenta a dificuldade da manutenção. Ou seja, a forma de reúso que se almeja é o de referência ao elemento reutilizado sem realizar qualquer alteração nele.

Idealmente, os elementos do conjunto de solução deveriam ser estritamente ortogonais, no sentido que cada um possua nenhuma superposição com qualquer outro elemento do conjunto. 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.

Cabe aqui salientar a diferença entre ortogonalidade e necessidade. Um elemento é necessário caso ele resolva uma parte da abstração raiz. Mas um subconjunto de elementos poderia resolver os mesmos itens da abstração raiz que outro subconjunto do mesmo conjunto de solução. Neste caso os subconjuntos em questão deixam de ser ortogonais. É evidente que existe redundância neste caso. Reorganizando e redefinido o conjunto de solução pode-se eliminar esta redundância, tornando ortogonal o conjunto. Em raros casos nos quais é necessário altíssimo desempenho, pode ser necessário criar redundâncias. Nestes casos deve-se anotar explicitamente a existência e o por quê desta redundância.

Exemplo

No exemplo do sistema de arquivos, a função *Criar arquivo* e a função *Abrir arquivo para gravação* não são estritamente ortogonais, uma vez que ambos levam à criação de um arquivo. De modo semelhante, as funções *Abrir arquivo para leitura* e *Abrir arquivo para gravação* não são estritamente ortogonais com a função *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 o de um *arquivo desconhecido*, troca o nome do arquivo para o segundo nome.
- *Verificar se arquivo existe* – dado um nome de arquivo, retorna TRUE se já for 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. Ao colocar em uso, é definido o modo de uso: *ler*, *gravar* ou *atualizar*.
- *Fechar arquivo* – dada a identificação de um arquivo em uso, retira o arquivo de uso.

Note que este conjunto será ortogonal, somente se arquivos a ler, arquivos a atualizar (ler e gravar) e arquivos a criar (somente gravar) possuírem direitos de acesso explicitamente diferentes.

A seguir ilustramos o uso destas funções com um trecho de programa esquemático no qual abrimos um arquivo para criar o seu conteúdo. Caso o arquivo a gravar já exista, o seu nome será trocado para BAK:

```
Se ExisteArquivo( {NomeArquivo, TXT} )
Então
     Se ExisteArquivo( {NomeArquivo, BAK} )
     Então
          DestruirArquivo( {NomeArquivo, BAK} )
     Fim se
     RebatizarArquivo( {NomeArquivo, TXT} ; {NomeArquivo, BAK} )
Fim se
CriarArquivo( {NomeArquivo, TXT} )
IdArqUso = AbrirArquivo( {NomeArquivo, TXT} ; GERAR )
```

3.8 Regras do critério viabilidade

Regra 24: Examine cada um dos elementos do conjunto de solução e determine se algum deles possui requisitos difíceis ou mesmo impossíveis de serem atingidos. Caso algum elemento seja inviável, a estrutura de solução precisa ser reformulada.

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 quando já se sabe que é impossível satisfazer objetivos e requisitos de algum dos elementos.

3.9 Regras do critério definição

Regra 25: Cada elemento deve possuir uma definição precisa de sua intenção.

Recom. 26: Utilize sempre uma forma de redação precisa.

Deve estar claro qual é a intenção de cada elemento. A intenção é identificada pelo nome dado ao elemento e por uma definição ou descrição. O nome do elemento deve ser uma linha de texto e não um nome de código C ou C++. O nome deve refletir bem o objetivo principal do elemento.

Caso a intenção do elemento seja complexa, não basta o nome para definir a intenção completa. Neste caso deve ser fornecida um texto definindo a intenção do elemento. Esta definição deve deixar clara a intenção do elemento, no entanto sem detalhar como realizar esta intenção. Procure sempre utilizar uma forma rigorosa de redigir. Na norma *PG-05 Regras e recomendações para a inclusão de especificações no código de programas* são apresentadas regras e recomendações de escrita.

A definição de um elemento composto pode apresentar diversas facetas. Por exemplo, a definição de um elemento poderia enumerar uma lista de objetivos a serem alcançados pelo elemento. Cada um destes objetivos corresponde a uma faceta.

Exemplo

Considere, por exemplo, a função *Ler registro*. Uma possível definição seria:

Ler registro Lê um registro de um arquivo determinado pelo descritor de arquivo em uso para um determinado *buffer* de leitura.

Note que nesta definição não se enumeram os parâmetros da função, nem se fala de tamanho de registro, tamanho de *buffer*, o que fazer se os dados estiverem errados e o que fazer em condições anormais de execução.

3.10 Regras do critério necessidade de item

Regra 27: Para cada um dos itens contidos na especificação da interface verifique se está relacionado com a intenção do elemento.

Regra 28: Todos os itens não relacionados com a intenção do elemento devem ser eliminados.

Recom. 29: Verifique se é possível reduzir a abrangência da intenção sem comprometer o papel do elemento no conjunto de solução.

Uma especificação é formada por um conjunto de itens. Alguns destes itens definem propriedades essenciais, outros definem critérios de aceitação da implementação do elemento. As propriedades essenciais descrevem a funcionalidade, ou o propósito do elemento sendo especificado. Os demais itens definem critérios de aceitação da solução – *requisitos*. Especificações podem basear-se em pressupostos – *hipóteses*. Finalmente, em alguns casos pode ser necessário estabelecer explicitamente *restrições* à forma de implementar o elemento.

É evidente que todos os itens de uma especificação devem contribuir de alguma forma para alcançar a intenção, ou *objetivo*, do elemento. Portanto, devem ser eliminados todos os itens para os quais não exista um vínculo claro com a intenção, ou para os quais não se possa produzir uma justificativa objetiva quanto à sua inclusão na especificação.

Evite especificar elementos estabelecendo uma enormidade de possibilidades, das quais somente algumas são utilizadas. Todas as características não utilizadas aumentam os custos, a dificuldade para o usuário ou o programador aprender a utilizar o componente, e o tempo necessitado para desenvolver. Pior, todas estas características são fontes de problemas quando acidentalmente utilizadas. Evite definir elementos contendo facetas de pouca utilidade em sua definição.

Com relação a funções, a especificação de uma função *F* deverá definir somente as propriedades de *F*, não devendo definir as propriedades das funções chamadas no corpo de *F*. Por exemplo, se uma função *G* chamada por *F* interage com o usuário, este fato não estará definido na função *F*.

Exemplo

Considere a especificação a seguir:

Ler registro
- lê *n* bytes a partir da posição corrente definida no descritor de arquivos
- avança a posição corrente de *n* bytes
- se, durante a leitura ou ao final dela, atingiu fim de arquivo sinaliza *EOF*

Neste exemplo o item *se atingiu fim de arquivo sinaliza EOF* é desnecessário, uma vez que existe uma função *Verificar se atingiu fim de arquivo* que deve ser ortogonal à função *Ler registro*.

3.11 Regras do critério suficiência de itens

Regra 30: Verifique se existem partes da intenção do elemento que não estejam sendo descritas pelos itens da especificação.

Regra 31: Verifique se podem ser adicionados itens à especificação, tornando mais completa a intenção do elemento

A especificação de cada elemento deve ser completa, tanto do ponto de vista da essência, como do ponto de vista dos critérios e aceitação. Por outro lado, ao examinar uma determinada especificação pode ser observado que uma pequena alteração da especificação torna o elemento sendo especificado mais abrangente, permitindo que seja efetivamente reutilizado em diversos lugares. No entanto, evite aumentar a especificação a menos que consiga justificar muito bem este acréscimo. Note que, aumentar a especificação, torna necessário rever todo o conjunto de solução.

Exemplo

Considere a especificação a seguir:

Ler registro
- lê *n* bytes a partir da posição corrente definida no descritor de arquivos
- avança a posição corrente de *n* bytes

Esta especificação é evidentemente incompleta. Como é que se sabe qual é o arquivo a ser lido? Lê para onde? O *descritor de arquivos em uso* contém a posição de leitura? O que fazer se o cursor de leitura estiver além do fim de arquivo? Como detectar e o que fazer se a leitura ler mais bytes do que existem no *buffer* de leitura? O que fazer ser ocorrer erro de leitura?

Considere uma nova especificação para a função *Ler registro*:

- Parâmetros de entrada:
  - hndArquivo — referência para o descritor de arquivo em uso. Identifica o arquivo físico, os direitos de acesso com que foi aberto, e a posição do cursor de leitura ou gravação.
  - Destino — estrutura informando o endereço do *buffer* de leitura, sua dimensão, e número de caracteres lidos.
  - NumBytes — número máximo de bytes que poderão ser lidos.
- Estado de entrada
  Caso o meio do arquivo físico não permita posicionamento explícito do cursor de leitura, assume-se que o arquivo físico esteja posicionado em um ponto consistente com a posição definida no descritor de arquivo em uso.
- Valor retornado
  condição de término da função, valores:
  - ARQ_OK — foram lidos exatamente NumBytes bytes.
  - ARQ_TAM — foram lidos menos do que NumBytes bytes.
  - ARQ_ERRO — ocorreu erro permanente de leitura. *Buffer* conterá a imagem do que foi efetivamente lido.
  - ARQ_PARM — parâmetros de entrada não valem: *hndArquivo* não é um *handle* válido, *Destino* não referencia um *buffer* válido, *NumBytes* é maior do que a dimensão do *buffer*, *NumBytes* é menor do que 1. *Buffer* permanece inalterado.
  - ARQ_FIM — ao entrar na função o cursor de leitura está após ao último byte do arquivo. *Buffer* permanece inalterado.
  - ARQ_ACESS — direitos de acesso não permitem ler. *Buffer* permanece inalterado.
  - ARQ_DISP — dispositivo não permite leitura. *Buffer* permanece inalterado.
- Valores retornados pelos parâmetros
  - Destino.Buffer — se foi lido alguma coisa, conterá os bytes lidos. Os bytes com índices maior ou igual a *NumLidos* permanecem inalterados.
  - Destino.NumLidos — se foi lido alguma coisa, o número de bytes lidos, senão conterá 0.
- Estado de saída
  Se foi lido alguma coisa, a posição do cursor de leitura estará imediatamente após ao último byte lido. Caso o meio do arquivo físico não permita posicionamento explícito do cursor de leitura (por exemplo

ao ler de teclado), o arquivo físico estará posicionado em ponto consistente com a posição definida no descritor de arquivo em uso.

3.12 Regras do critério estrutura de itens

Regra 32: Agregue os itens da especificação formando estruturas de itens.

Os itens de especificação devem ser apresentados de forma estruturada. A estrutura de itens pode afetar somente a organização da especificação, como pode afetar as estruturas de dados, funções e classes utilizadas na especificação. O exemplo acima ilustra uma forma de se estruturar os itens de especificação. A norma *PG-05 Regras e recomendações para a inclusão de especificações no código de programas* define como organizar e estruturar as especificações incluídas no código.

Exemplo

No exemplo acima cabe salientar a agregação realizada no tipo do parâmetro *Destino*. Este tipo contém todas as informações manipuladas ao preencher um buffer. Em C++ *Destino* será tipicamente um objeto. Em C a tendência é passar todos os elementos de *Destino* como parâmetros individuais. Aplicando-se a regra com rigor, os itens que compõem *Destino* devem ser agregados em um `struct`.

Muitas vezes existem diversas formas de agregar, todas elas válidas segundo a regra de estrutura de itens. Neste caso deve-se optar pela forma que parece ser mais conveniente. No exemplo acima, agregamos o número de bytes lidos no tipo do parâmetro *Destino*. Ao invés disso poderíamos ter agregado em um `struct` o número de bytes lidos com a condição de retorno. Neste caso a função retornaria um valor composto. Esta forma de agregar dificulta um pouco o tratamento da condição de retorno a ser realizado nas funções cliente.

3.13 Regras do critério ortogonalidade dos itens

Regra 33: Assegure que cada item da especificação aborde uma faceta da intenção do elemento, ou um critério de aceitação, que não seja abordado por nenhum outro item dessa especificação.

Os itens de uma especificação não devem ter superposições. Caso o tenham, certamente existirá redundância no conjunto de itens. Esta redundância freqüentemente resulta em especificações inconsistentes ou contraditórias.

Exemplo

Na função *Ler registro* descrita acima, tomamos o cuidado de definir o efeito da leitura sobre o parâmetro *Destino* sem nos referirmos explicitamente aos valores retornados pela função. Já na seção de valores retornados pela função, explicamos o significado de cada um, sem especificar o efeito no parâmetro *Destino*.

3.14 Regras do critério encapsulamento

Regra 34: Torne públicos somente os itens que precisam ser do conhecimento do cliente ou do usuário do elemento sendo especificado.

Recom. 35: Assegure que a abrangência ou o escopo de um item de especificação seja o menor possível.

Especificações contém itens públicos e encapsulados. Os itens públicos são utilizados por:

- projetistas e programadores ao utilizar o elemento como parte de uma solução.
- usuários de programas ao interagirem com o elemento sendo especificado.

Quanto menos itens públicos existirem, mais fácil será entender como utilizar o elemento. Somente torne um item público se for efetivamente necessário.

Em uma estrutura podemos escolher diversos pontos onde especificar uma propriedade. Procure sempre o nível de menor abstração que abranja todos os elementos afetados pelo item. Se for possível, particione a especificação em uma parte abrangente e uma outra detalhada.

A dificuldade maior de se obedecer a estas regras reside no particionamento de especificações em dois documentos, um público destinado a possíveis desenvolvedores usuários do elemento, e o outro destinado aos desenvolvedores do elemento. Esta dificuldade pode ser vencida com o uso de ferramentas de geração de documentação.

Exemplo

No módulo de processamento de arquivos, o tipo de dados que informa as condições de retorno deve ser particionado em 3 grupos. Um grupo tratando dos erros a nível de sistema de arquivos. Outro, tratando de erros a nível de arquivos em uso. Finalmente, o terceiro tratando de erros a nível de registros. Em adição, a cada nível definem-se os valores que podem ser retornados de forma genérica, detalhando o significado em cada uma das funções. Por exemplo, o valor ARQ_ERRO indica que ocorreu um erro permanente. Este significado do valor é genérico. Ao ler, terão sido lidos alguns bytes, permitindo recuperar parcialmente a informação contida no arquivo. Ao gravar, nada deverá ter sido gravado, evitando que se tenha valores duvidosos gravados em arquivos. Estes dois últimos significados são específicos de cada função, devendo, pois, estar incluído nas respectivas especificações.

Em C++ variáveis locais podem ser declaradas no interior de blocos. Utilize esta possibilidade para declarar variáveis com o menor escopo possível. Isto reduz possíveis interferências entre usos de variáveis, veja a norma: *PG-03 Regras e recomendações para programação em C e C++*.

3.15 Regras do critério flexibilidade

Recom. 36: Procure especificar elementos capazes de serem adaptados a uma gama grande de diferentes condições de uso.

Sempre que for possível, procure especificar elementos de modo que possam ser adaptados a diferentes condições de uso. Isto pode ser conseguido por exemplo através do uso de

parâmetros exemplo: ao invés de ler para um *buffer* de tamanho fixo, permita que se informe o tamanho do *buffer*. Imediatamente a função poderá ser utilizada em vários contextos diferentes.

estados internos exemplo: ao invés de abrir um arquivo para ler, ou para gravar utilize um parâmetro que informa o direito de acesso pretendido. Este direito de acesso será guardado no estado do arquivo em uso e será consultado ao autorizar operações sobre registros.

interpretação pode-se criar uma linguagem a ser utilizada para problemas semelhantes. Cada caso específico é agora definido nesta linguagem e é interpretado pelo elemento. Exemplos são os diversos processadores de menus e diálogos existentes no mercado.

Esta recomendação pode conflitar com as regras do item *Regras do critério necessidade de item*. Em caso de conflito opte sempre pela simplificação da especificação.

Bibliografia

[Humphrey 90] Humphrey, W.S.; *Managing the Software Process*; Addison Wesley; 1990

[Humphrey 95] Humphrey, W.S.; *Personal Software Process*; Addison Wesley; 1995

[Mafeo 92] Maffeo, B.; *Engenharia de Software e Especificação de Sistemas*; Rio de Janeiro; Campus; 1992

[McMenamin 84] McMenamin, S.M.; Palmer, J.F.; *Essential Systems Analysis*; Yourdon Press; 1984

[Page-Jones 80] Page-Jones, L.; *The Practical Guide to Structured System Design*; Englewood Cliffs, NJ; Prentice Hall; 1980

[Paulk 93] Paulk et al; *Capability Maturity Model for Software*, version 1.1, Technical Reports SEI-93-TR-24; Software Engineering Institute, Carnegie Mellon University; 1993

[Riding 92] 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

[Rocha 87] Rocha, A.R.C.; *Análise e Projeto Estruturados de Sistemas*; Livros Técnicos e Científicos; Rio de Janeiro; 1987