Aula 10

Padrões de Programação 2

"Mais regras?!" ALUNO SUPER-ESPANTADO

"A forma mais rápida de desenvolver um novo hábito é não ter outra alternativa" [Hadden 1999]

Alessandro Garcia
LES/DI/PUC-Rio
Abril 2010

Anúncios

- No fim da aula: anotar dúvidas relacionadas ao trabalho 1 e entregar
- Enunciado do trabalho 2 será divulgado esta semana
- Lembrete, prova 1: **3 de maio**
  - vocês gostariam de receber exemplos de questões de provas anteriores?

Aula Passada: revisando conceitos...

- O que é análise estática? Exemplos de técnicas de análise estática?
- Exemplos clássicos de padrões de falta?
- Por que detecção (de violações) e/ou aplicação de padrões de falta é complementar as atividades de teste?

Hoje... especificação

- Objetivo dessa aula
  - Entender a organização de catálogos de padrões de programação
  - Entender a importância de padrões de documentação, estilo, estruturação de módulos, entre outros
  - Descrever estrutura do catálogo de padrões em [Staa00]
  - Realizar exercício para identificar violações a padrões "universais" de estruturação e documentação (e de faltas)
- Referência básica:
  - Capítulo 4 e apêndices do livro

Exemplos de Falhas Introduzidas...

- ... Por programadores experientes e "gurus"
- ... Presentes em sistemas de software reais, amplamente utilizados

"Using Static Analysis for Software Defect Detection"

http://www.youtube.com/watch?v=GgK20Yv9QRk

Google TechTalks
July 6, 2006
William Pugh
Uso e desenvolvimento da ferramenta: FindBugs

Google engEDU
Tech Talks Channel

Documentando Padrões de Faltas...

- Padrões de Faltas... (*Bug Patterns*)
  - É importante mudar nossos processos de desenvolvimento de forma aprendermos com nossos próprios erros
  - A cada defeito encontrado, verificar se o mesmo pode ser transformado em um **padrão de falta**
    - muitas falhas se manifestam recorrentemente em um projeto
    - catálogos universais de padrões não são suficientes
      - muitos faltas são parte da cultura organizacional
      - membros de um time de desenvolvimento tendem a cometer as mesmas falhas específicas
  - **Atalho** - nos permite de forma mais efetiva:
    - Implementar detectores de 'bugs'
    - Definir casos de teste
      - definir casos de teste efetivos é caro!

Necessidade de um catálogo para inspeção

- **Variáveis**
  - Todas variáveis estão sendo adequadamente inicializadas?
  - Existem variáveis com nomes similares/confusos?
- **Funções (ou Métodos)**
  - O valor de cada argumento (parâmetro) testado antes de ser usado?
    - Por exemplo, valores "null"
- **Operadores relacionais (ou de comparação)**
  - O uso dos operadores de comparação está correto?
  - Toda expressão está correta?
- **Fluxo correto**
  - Todos os laços (*loops*) terminam?

Regras de Inspeção (Natureza da Falta)

Faltas de Dados

Todas as constantes foram nomeadas (apropriadamente)?

Todas variáveis foram inicializadas antes dos respectivos valores serem utilizados?

Caso strings de caracteres sejam usados, um delimitador está sendo utilizado (ex.: EOF)?

O limite superior de arrays deve ser igual ao tamanho do array ou tamanho – 1?

Faltas de Controle

Para cada expressão condicional, a condição está correta?

Todos os laços estão terminando de forma apropriada?

As expressões compostas fazem uso correto de parênteses?

Em blocos "case", todos os possíveis casos estão sendo explicitamente tratados?

Se "break" é necessário no fim de cada bloco "case", tal "break" foi realmente incluído?

Regras de Inspeção

Laboratório de Engenharia de Software

Faltas de Entrada e Saída

Algum valor de entrada não esperado pode causar problemas?
Todas as variáveis de entrada em parâmetros são usadas corretamente?
Todas as variáveis de saída são atribuídas um valor localmente antes de serem retornadas?

Faltas de Exceções

Todas as condições excepcionais (ex.: arquivo não existente) são consideradas?

Regras de Inspeção

Laboratório de Engenharia de Software

Faltas de Interface

Todas chamadas de funções ou métodos têm parâmetros na ordem correta?
Se módulos compartilham espaços de memória, todos possuem o mesmo modelo da estrutura da memória compartilhada?

Falhas de Armazenamento

Se uma estrutura de dados ligada é modificada, todos os ponteiros foram corretamente atualizados?
Se armazenamento **dinâmico** é usado, o **espaço** necessário foi **alocado** corretamente?
O **espaço** alocado é explicitamente **desalocado** na medida que não é mais necessário?

Padrões de Estruturação e Documentação

- Por que estruturar e documentar bem o código?
  - facilitar a **reutilização** e **manutenção** do código-fonte
    - fluxo de programadores é alto e constante
  - disponibilizar **documentação profissional** de bibliotecas para usuários do software: ***documentação da interface***
  - disponibilizar **documentação profissional** da implementação interna do módulo: ***documentação da implementação***
  - contribuir com a ***gestão do conhecimento*** relativos aos projetos de software de uma organização

- Atividade de leitura de código é, de longe, a mais comum em desenvolvimento e manutenção de software

O que este módulo (em Java) faz?


```
public class do {
   public static final String m = "E";
   private int s;

   public void de ( int i ) {
      s += i;
   }

   public void sa ( int i ) {
      s -= i;
   }

   public String get() {
      return Integer.toString( s ) + " "  + m;
   }
}
```

Este código tem algum significado para você?

Padrões Universais: Política de Nomes

- Existem convenções de nomes padrões para identificadores:
  - Módulos (e/ou classes)
    - Devem representar "coisas" do mundo real, seja entidades conceituais (p.e. Curso) ou físicas (e.g. Pessoa)
      - ou estruturas de dados: árvores, pilhas, etc...
    - Nomes de módulos deveriam ser **substantivos**
    - Use nomes descritivos e simples, evitando abreviações na medida do possível
      - Exceção: siglas/abreviações conhecidas na organização
    - Nomes deveriam conter maiúsculas…
      - … e minúsculas, com a primeira letra de cada palavra sendo maiúscula, p.e. ContaBancária
      - … somente [Staa00]: p.e. CONTABANCÁRIA
      - Obs.: alguns padrões sugerem o uso de _ "underscore" para separação de palavras

Vide livro texto [Staa00] para um esquema mais sofisticado de políticas de nomes

Padrões Universais: Política de Nomes

- Funções (e/ou Métodos)
  - funções tipicamente representam ações a serem realizadas por um módulo/objeto (p.e. adicionarAluno no módulo Curso)
    - Nomes de funções deveriam ser **verbos**
  - nomes de funções (métodos) deveriam ter maiúsculas e minúsculas, com a 1ª. Letra de cada palavra interna como maiúscula:
    - primeira letra maiúscula [Staa00] ou minúscula: AdicionarAluno() ou adicionarAluno()
      - » em [Staa00] também sugere-se uso de prefixo do módulo para funções públicas: CUR_AdicionarAluno()

Vide livro texto [Staa00] para um esquema mais sofisticado de políticas de nomes

Padrões Universais: Política de Nomes

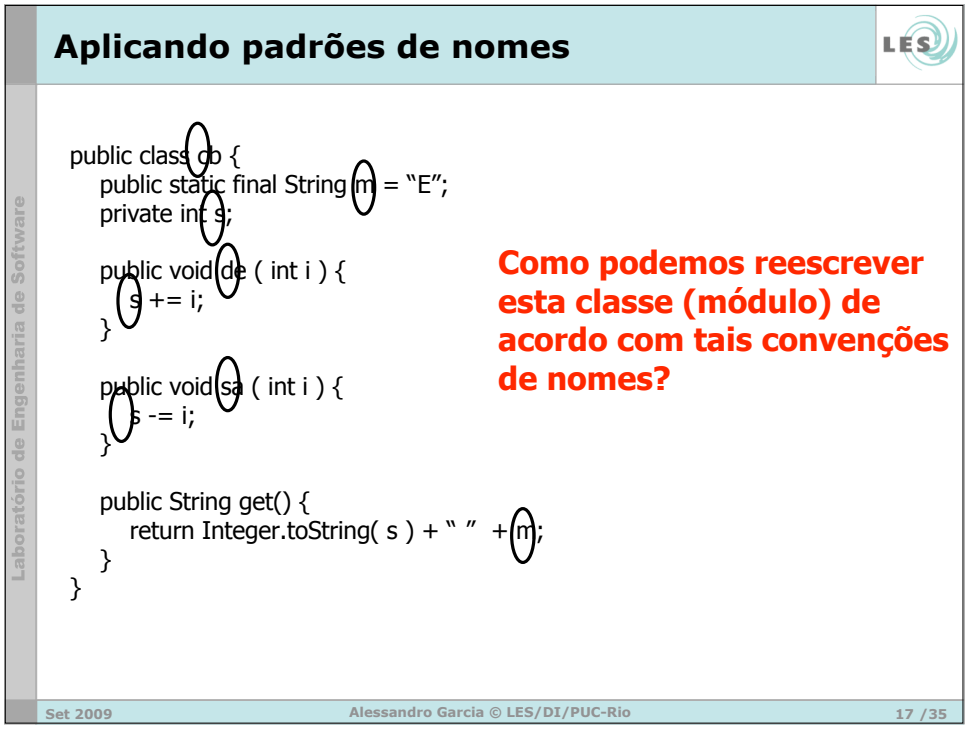
- Variáveis
  - Nomes deveriam ser curtos e com significado óbvio
  - *Nomes de variáveis com um caracter:* somente para variáveis temporárias de escopo interno à uma função
    - Exemplo: variável de índice declarada em um **for/while**
  - Nomes de variáveis deveriam começar com minúsculas, com cada inicial de palavra interna em maiúscula
    - Exemplo: idadeAluno
    - para os casos de variáveis com escopo "público", deve-se começar com a primeira letra maiúscula [Staa00]
      - » variáveis em parâmetros e condições de retorno

Vide livro texto [Staa00] para um esquema mais sofisticado de políticas de nomes

Padrões Universais: Política de Nomes

- Constantes
  - Os nomes de constantes deveriam estar completamente formados por maiúsculas, com um *underscore* separando as palavras
    - Exemplo: NULL
    - Exemplo: NÚMERO_MÁXIMO_DE_ALUNOS
  - Sufixos e prefixos podem ser utilizados:
    - CRIAR_ARV_CMD
    - INS_DIR_CMD
    - INS_ESQ_CMD
    - IR_PAI_CMD
    - IR_ESQ_CMD
    - IR_DIR_CMD
    - OBTER_VAL_CMD
    - DESTROI_CMD

Aplicando padrões de nomes



```
public class cb {
    public static final String m = "E";
    private int s;

    public void de ( int i ) {
        s += i;
    }

    public void sa ( int i ) {
        s -= i;
    }

    public String get() {
        return Integer.toString( s ) + " "  + m;
    }
}
```

Como podemos reescrever esta classe (módulo) de acordo com tais convenções de nomes?

Aplicando padrões de nomes

```
public class ContaBancária {
    public static final String MOEDA = "Euros";
    private int saldo;

    public void deposito( int quantia ) {
        saldo += quantia;
    }

    public void saque( int quantia ) {
        saldo -= quantia;
    }

    public String obterSaldo() {
        return Integer.toString( saldo ) + " " + MOEDA ;
    }
}
```

Reescrever o módulo de acordo com os padrões de nomes leva à um texto de módulo que é muito mais fácil de ler, manter e modificar.

Regras "universais": endentação

- Endentação
  - Use approx. 4 espaços de endentação para cada bloco de código
  - **Evite linhas** muito **longas**
    - Ideal: não muito mais que 80 caracteres
- Padrões de quebra de linha que continuam anteriores
  - Endente a segunda linha para indicar relacionamento com a linha de cima

```
someFunction(longExpression1, longExpression2, longExpression3,
                longExpression4, longExpression5);

longName1 = longName2 * (longName3 + longName4 - longName5)
                + 4 * longname6;
```

Vide livro texto [Staa00] para um esquema mais sofisticado de regras ou recomendações para endentação

Regras "universais": declarações de variáveis

- Declarações
  - Declare cada variável em uma linha separada
  - Inicialize variáveis na medida que você as declara
  - Coloque declarações somente no começo de um bloco
    - Um bloco é uma estrutura cercada de { e }

```
public boolean addStudent( String name ) {
   boolean success = false;    // success é o valor de retorno deste método

   if( numberOfStudents < MAXIMUM_NUMBER_OF_STUDENTS ) {
      int studentPosition = getStudentPosition( name );
      ...
   }
   ...
}
```

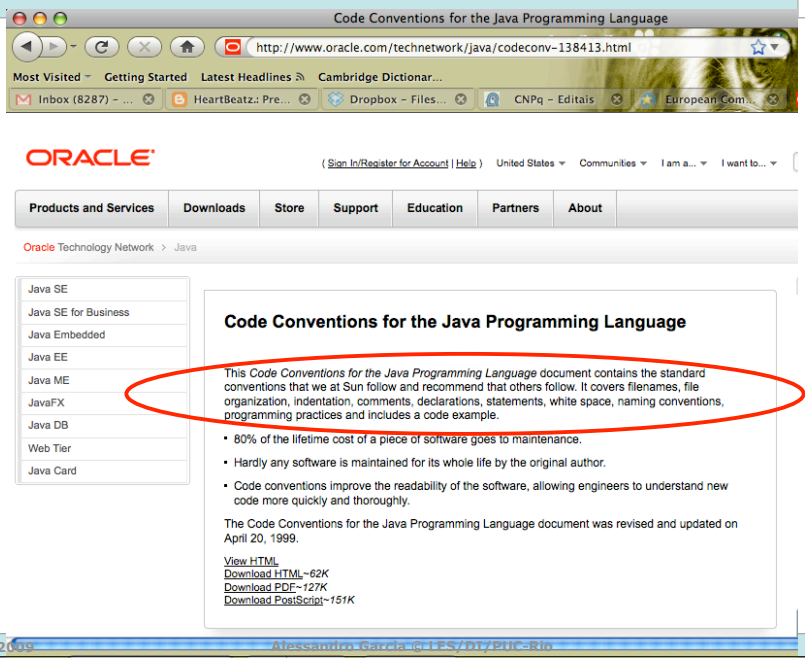
Regras "universais": declarações de variáveis

- Cada linha de comando deveria estar em uma única linha
  - Coloque cada comando (p.e. atribuição de variável) em uma linha separada
    - index = 0; match = false; ✗
- Operadores binários
  - Espaços em branco entre operadores

```
a += c + d;
a = (a + b) / (c * d);
```

  - Exceto: ++, --, += etc

Catálogos públicos (na WEB)



Elementos desejáveis na descrição de um padrão

- Descrição sucinta e sem ambigüidades da ordem ou recomendação
  - pode ser uma afirmação (ordem) ou uma pergunta
- Contexto: código fonte vs. outros artefatos, desenvolvimento vs. manutenção vs. revisão, etc...
- Variantes: soluções alternativas
  - exemplo: possíveis soluções dadas ao problema da divisão de ponteiros na aula passada
- Exceções: alternativas ou casos onde a solução não precisa ou não deve ser aplicada
  - exemplo: funções encapsuladas não conterão o prefixo do módulo [Staa00, pg. 640]
- Elementos opcionais, mas desejáveis:
  - razão da existência da regra ou recomendação
  - exemplo: antes vs. depois

Tipos de Padrões – Catálogo [Staa00]

- Padrões de faltas (aula passada)
  - Objetivo: reduzir a freqüência de faltas mais corriqueiras
  - Fontes: alguns descritos no (Apêndice 3, Staa00) e artigo "C Traps and Pitfalls)
- Padrão (de composição) de módulos (Apêndice 1, Staa00)
  - Objetivos:
    - assegurar a existência de definições de interfaces entre módulos
    - assegurar a consistência dos módulos que compartilhem uma mesma interface
  - Exemplo de regras:
    - ***Regra 2***: cada módulo será composto por dois arquivos
      - um contendo o módulo de definição (interface explícita)
      - outro contendo o correspondente módulo de implementação

Padrão de Módulos (Apêndice 1, Staa00)
Estrutura de módulo de definição


```
/*****************************************************************
/*
/*  Módulo de definição: Módulo exemplo
/*
/*  Nome do arquivo:     EXEMP.H
/*
/*****************************************************************/
#if !defined( EXEMP_MOD )
#define EXEMP_MOD

   ...



/***** Tipo de dados exportado pelo módulo *****/
     typedef struct
     {
        int UmInt ;
        int OutroInt ;
     } EX_tpMeuTipo ;


#endif
/********** Fim da definição: modulo **********/
```

Recomendação 9: cabeçalho do módulo

Regra 4: controle para evitar a inclusão duplicada ao compilar um módulo.

Regras 6, 7: declare os tipos/estruturas exportados; incialize as estruturas

Padrão de Módulos (Apêndice 1, Staa00)
Estrutura de módulo de implementação


```
/*****************************************************************
/*
/*  Módulo de implementação: Módulo exemplo
/*
/*  Nome do arquivo:     EXEMP.C
/*
/*****************************************************************/

/* Inclusões do compilador /

#include <stdio.h>

/* Inclusão do respectivo módulo de definição /

#include  "EXEMP.H"

/* Inclusão de módulos de definição de servidores */

#include  "MODULO1.H"
#include  "MODULO2.H"

/* Código do módulo de implementação */
...
/********** Fim da implementação: modulo **********/
```

Recomendação 9: cabeçalho do módulo

Regra 5: inclusões requeridas pelas bibliotecas do compilador

Regra 13: redija o código de inclusão do respectivo módulo de definição

Regra 9: definição dos módulos servidores

Tipos de Padrões

- Padrões de especificações (Apêndice 2, Staa00)
  - objetivos:
    - assegurar a existência e facilitar a manutenção de informações gerenciais relativas aos módulos e demais arquivos do programa
    - assegurar estilo de documentação consistente, uniforme e independente de autor
    - facilitar a modificação de programas redigidos por outros
  - Exemplo de regra - comentário:
    - ***Regra 6***: utilize o esquema de código a seguir para tornar temporariamente não compilável uma parte programa

```
#ifdef 0
    ...
#endif
```


Tipos de Padrões

- Padrões de especificações
  - Informações gerencias importantes
    - Identificação do arquivo: nome, proprietário, projeto, gestor, versão corrente, *data de aprovação*
    - Descrição do conteúdo
    - Histórico de alterações: versão, data, autores, solicitações
- Padrões de estilo (Apêndice 4, Staa00)
  - Objetivos:
    - uniformizar o estilo de programação da organização
    - facilitar a modificação de programas redigidos por outros
  - Exemplo de regra:
    - ***Regra 13***: declare cada variável em linha individual

Tipos de Padrões

- Outros tipos de padrões
  - Padrões para escolha de nomes (Apêndice 5, Staa00)
  - Padrões para constantes simbólicas (Apêndice 6, Staa00)
  - Padrões para uso de ponteiros (Apêndice 7, Staa00)
  - Padrões específicos para C++ (Apêndice 8, Staa00)
  - Recomendações para testes de módulos (Apêndice 10, Staa00)
- Refatorações (regras para reorganizar código) também podem ser vistos como padrões, mas eles implicam em uma transformação do programa
  - sem modificar sua semântica

Adoção de Padrões

- Três fases:
  - Exposição dos padrões aos membros do time
    - Antes de entrar em vigor
  - Treinamento dos membros
    - Muitos padrões requerem conhecimento detalhado das linguagens e métodos relacionados (p.e. linguagem de programação C)
    - Essencial para o êxito da adoção
  - Controle da adoção
    - Implementação de ferramentas ou uso de ambientes de desenvolvimento
      - ferramentas de detecção de padrões de faltas
      - ferramentas de documentação (JavaDoc, Doxygen, etc...)
    - Controlador de qualidade

Indo Adiante...

- Entrevistas no Rádio Engenharia de Software (em inglês)
  - Software Engineering Radio:
    http://www.se-radio.net/
    - Episódios 110 e 112: Papéis do Engenheiro de Software
      - O que é esperado de gerentes de projetos com qualidade, líderes técnicos, testadores, desenvolvedores júniores, etc...
    - Outros Episódios: Refatoração (46, 55), Contratos (51),

Aula 10

Padrões de Programação 2

"Mais regras?!" ALUNO SUPER-ESPANTADO

"A forma mais rápida de desenvolver um novo hábito é não ter outra alternativa" [Hadden 1999]

Alessandro Garcia
LES/DI/PUC-Rio
Abril 2010