Engenharia de Software 2 - Aula 01
Qualidade de Código

Referência:
R. Pressman and B. Maxim, Software Engineering: A Practitioner's Approach, 8^{th} edition, McGrall-Hill Education, 2015 [1]

"Don't aspire to be the best on the team, aspire to be the best for the team." — Anonymous

1 Preparação

- A sala será dividida em equipes com 5 ou 6 pessoas. As equipes serão sorteadas pelo Moodle.
- Cada equipe terá um Líder. O líder será indicado antecipadamente.
- O líder é responsável por coordenar o trabalho da equipe para atender o que foi solicitado na atividade da aula, preparar o relatório e entregar dentro do prazo estipulado.

2 Sistema de Escolha de Times de Vôlei

2.1 Requisitos

Suponha que o professor da disciplina seja um jogador de vôlei de areia amador. Sendo ele um entusiasta do esporte montou e coordena um grupo de jogadores que se encontra regularmente para jogar. Com a popularidade alcançada por suas publicações em redes sociais o grupo de jogadores vem aumentando. Chegando a quase 20 jogadores em algumas ocasiões.

O professor gostaria de desenvolver um software para dividir os jogadores em equipes. O software deve atender aos seguintes requisitos:

1. O software deve permitir o cadastro dos jogadores, incluindo seu nome e o seu ranking.
2. O ranking é um valor inteiro que classifica o jogador em um certo nível de habilidade. Para facilitar considerem que valores mais baixos indicam que o jogador é melhor.
3. Um jogador cadastrado pode ou não estar presente em um determinado jogo. Somente jogadores que estiverem presentes devem ser considerados na formação dos times.
4. O número de times que será montado é uma entrada do programa. Ou seja, o usuário indica quantos times ele quer.
5. A montagem dos times deve seguir a regra: Para n times, os n melhores jogadores são selecionados inicialmente e cada um é colocado em um time. A partir daí os outros jogadores são colocados um em cada time, seguindo a ordem do seu ranking, mas em ordem inversa do time dos melhores jogadores. Ou seja, o melhor jogador depois do n é colocado no time do n, o próximo no $n-1$ e

assim por diante até que o jogador 1 receba mais um integrante para o time dele. A partir daí o próximo jogador é colocado novamente no time do jogador n e o processo se repete.

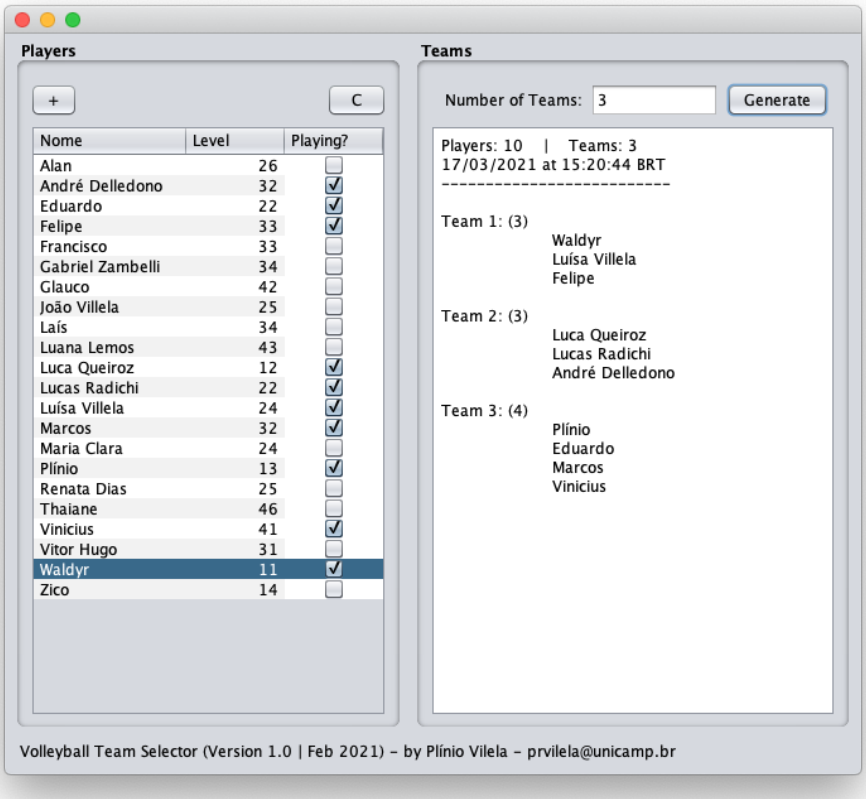

Figura 1: Exemplo de Execução

A Figura 1 apresenta um exemplo de execução do sistema, observe que do lado esquerdo estão os jogadores cadastrados, seus níveis definidos e a indicação de se eles vão jogar ou não. Do lado direito foi solicitado 3 times e a distribuição dos times é apresentada.

2.2 Execução e Teste da Implementação

Foi feita uma implementação desse sistema em Linguagem C, essa implementação está disponível para os grupos no Moodle. Faça o download do código, compile e execute.

Desenvolva alguns casos de teste para testar o programa. Execute esses casos de teste e anote as não conformidade.

3 Identificação das Violações de Qualidade do Código

Para esta segunda parte da atividade leia as notas de aula sobre Qualidade de Código do Prof. Fábio Levy Siqueira, considerando as recomendações para nomes, formatação e comentários apresentadas a partir da página 3 do PDF anexo.

Façam uma lista numerada das violações encontradas, utilize a numeração das linhas da listagem do código que se encontra ao final deste documento.

4 Entrega

A entrega de hoje deve incluir:

- O nome do grupo e os nomes de todos os integrantes.
- O plano de testes elaborado para o sistema, incluindo evidências de que os testes foram efetivamente executados (prints de tela).
- O relatório de violações de qualidade de código, as violações devem ser listadas e numeradas.
- Caso tenha sido encontrada alguma não conformidade durante os testes o defeito deve ser indicado.
- O código gerado para resolver as não conformidades e as violações de qualidade de código.
- Os líderes das equipes devem indicar no relatório se algum membro não participou das atividades.
- Ao final da aula uma das equipes será escolhida para fazer uma apresentação dos resultados obtidos.

Referências

[1] R. Pressman and B. Maxim. *Software Engineering: A Practitioner's Approach.* McGrall-Hill Education, 8 edition, 2015.

Qualidade de código

Autor: Fábio Levy Siqueira 27/03/2018

O código fonte é usado pelo compilador para gerar um executável (ou, no caso do Java, um *bytecode*). Para que o compilador possa fazer o seu trabalho é necessário que o código seja escrito seguindo algumas regras básicas. Por exemplo, o nome de uma variável não pode conter espaços e não pode começar com um número; uma variável deve ser declarada antes de ela ser usada; etc. O compilador verifica essas regras e, caso alguma delas não seja seguida, ele apresenta um *erro de compilação* e simplesmente rejeita o código. O desenvolvedor precisa corrigir o código para que ele atenda as regras básicas definidas pelo compilador.

Um código que compila não é necessariamente um bom código. Na realidade, compilar é o mínimo - jamais entregue um código que não compila! Mas então o que é um *bom código*?

Muitos programadores iniciantes acham que um bom código é aquele que é **correto**, ou seja, faz o que tem que fazer. Alguns programadores com mais experiência dirão que isso é insuficiente: um bom código também tem que ser **eficiente** no uso de recursos (usar pouca memória, demandar pouco tempo de processamento e usar pouca banda de rede). Mas o código ser correto e eficiente são *apenas* condições necessárias para um bom código. Elas seriam *necessárias e suficientes* se o código, depois de compilado, nunca mais fosse lido ou alterado. No mundo real, isso é extremamente improvável. É natural que um software sofra algum tipo de manutenção e, portanto, alguém algum dia terá que olhar novamente para o código para fazer alguma alteração nele. Colocando alguns números, de 40% a 70% dos custos do software são durante a sua manutenção (GRUBB; TAKANG, 2003, p. 6) e por volta de 50% do esforço da manutenção é entender o software a ser modificado (PARIKH; ZVEGINTZOC, 1983 apud PFLEEGER; ATLEE, 2009, p. 546). Um dos principais motivos para isso é que quem faz a manutenção muitas vezes não é a mesma pessoa que fez o código original. Mesmo se for, alguns meses ou anos de diferença fazem com que o desenvolver esqueça de muitos dos detalhes do código. Experimente olhar um código complexo que você fez no ano passado. Você provavelmente descobrirá que o seu código não é muito fácil de entender!

Portanto, um bom código precisa ser facilmente entendido por um outro desenvolvedor – ou seja, ele precisa ser **elegante**. O desagradável é que um desenvolvedor não pode ser tão exigente quanto um compilador e *rejeitar* um código mal escrito. Muitas vezes é necessário fazer uma correção urgente de um código que está em produção (ou seja, em um software sendo usado). Então se o código não seguir os padrões de qualidade do desenvolvedor, bem... *azar o dele*! O desenvolvedor terá que se esforçar para tentar entender o código – o que às vezes significa *refatora-lo* para torná-lo elegante, ou seja, alterar o software de forma que ele não altere seu comportamento externo, mas que melhore a sua estrutura interna (FOWLER, 1999). Assim sendo, um bom desenvolvedor sempre deve buscar escrever um código elegante para facilitar o trabalho de quem irá olhar o código depois dele. Portanto, não escreva o código para o compilador; escreva o código para que uma outra pessoa consiga ler. Quem já fez manutenção de um código escrito por um terceiro sabe o quão irritante, revoltante e trabalhoso é ter que entender um código mal escrito.

Ainda pensando na manutenção, uma outra característica importante de um bom código é que ele seja **testável**. Ou seja, deve ser fácil procurar defeitos no código.

Um código ruim

Se você acha que legibilidade é uma característica pouco importante para um bom software, veja o exemplo a seguir. O que esse código faz?

```
1  public static int m(int[] v) {
2    int i, j;
3    int k; int l , m;
4     k = v.length;
5     l = 0;
6    for (j =    0; j<k; j ++) {
7  if (v[j]>  0) {
8             m = v[j];
9    l = l + m;
10 }} return l;
11 }
```

Apesar de o código ser curto e fazer algo simples, não é imediato descobrir o que ele faz. Talvez o principal motivo para isso seja o uso de nomes pouco indicativos, como "m", "i", "j" e "k". Apesar de serem nomes curtos, eles não ajudam a entender para o quê a variável serve. Por mais que seja um costume na área de computação usar nomes desse tipo (um *mau* costume, por sinal), um "i" pode significar qualquer coisa – desde um controlador de `for` a até o menor número de uma sequência. Uma outra razão para ser difícil entender o código acima é sua *formatação*. Em Java os espaços, tabulações e linhas em branco não afetam os comandos – claro, contanto que não quebrem palavras. O importante para o compilador são símbolos como ";" "{" e "}" (isso é diferente na linguagem Python, por exemplo). Apesar disso, uma boa formatação é essencial para identificar se um comando está dentro de um laço ou de uma condição. No exemplo, não é claro se o "l = l + m;" da linha 9 está dentro do `for` ou do `if`. Por fim, um outro motivo para esse código ser difícil de entender é que ele possui comandos desnecessários. Três variáveis não são necessárias para o código funcionar: o "i", "k" e o "m". Colocá-las não torna o código mais fácil de ler, mas aumenta a complexidade dele e desperdiça recursos computacionais.

Um código bom

Antes de apresentar o que o código faz, a seguir é apresentado o mesmo código escrito de uma forma mais elegante.

```
1 public static int somaOsPositivos (int[] arranjo) {
2   int somaDosPositivos = 0;
3
4   for (int i = 0; i < arranjo.length; i++) {
5     if (arranjo[i] > 0) somaDosPositivos += arranjo[i];
6   }
7
8   return somaDosPositivos;
9 }
```

O código acima deixa claro o que ele faz: retorna a soma dos números positivos do arranjo passado como parâmetro. Tudo bem que o nome do método já esclarece isso, mas não é só isso que faz com que esse código seja melhor que a versão anterior dele. O nome das variáveis é sugestivo; a formatação evidencia o que está dentro e fora dos laços e condições; e não há nada atrapalhando o entendimento do código.

Recomendações

Escrever um bom código envolve seguir algumas regras, um pouco de inspiração (para encontrar boas soluções e escolher bons nomes) e um pouco de bom senso. A inspiração pode não aparecer (mesmo depois de você tomar um café) e o bom senso pode ser discutível, mas você não tem desculpa para deixar de seguir algumas regras. Algumas delas são tão básicas que um bom ambiente integrado de desenvolvimento (IDE) ajuda o programador a segui-las.

A seguir são apresentadas algumas regras básicas.

Nomes

Alguns programadores podem até argumentar que nomes menores são mais práticos pois facilitam a digitação. Mas isso não é uma desculpa razoável atualmente: qualquer IDE permite auto completar o código. Os nomes de variáveis e de outras informações (métodos e classes, que veremos adiante) devem ser *acima de tudo* representativos. Ou seja, se o "i" representa o menor número da lista, então chame-o de "menorNumeroDaLista".

Em Java se usa normalmente a convenção *Camel Case*. Nomes de variáveis seguem a convenção *lower Camel Case*: a primeira palavra do nome da variável deve ser escrita em letras minúsculas e as demais palavras devem ter a primeira letra em maiúscula e as demais minúsculas. Como exemplo, os seguintes nomes são corretos: "menor", "menorNumero", "menorNumeroDaLista", "funcionarios", "listaDeFuncionarios". Por outro lado, os seguintes nomes não estão corretos: "MenorNumero", "_menor", "Nome" e "nome_Real".

Convenção de código

As convenções de código definem desde o formato dos nomes, por exemplo, se devem ser separados por "_" ou usando Camel Case; o número de espaços para indentação; onde deve ser colocado os símbolos de abertura e fechamento de bloco ("{" e "}"); etc. Bons IDEs ajudam a seguir essas convenções, formatando o código enquanto o programador digita ou mesmo para reformatar o código inteiro automaticamente.

Existem inúmeras convenções de código. No Java os próprios criadores da linguagem definiram também a convenção de código a ser usada (SUN MICROSYSTEMS, 1997).

Na prática, desenvolvedores devem seguir a convenção usada pela organização que ele trabalha. Seguir uma convenção é fundamental para manter a base de código homogênea e facilitar a leitura do código por outros funcionários. Como um professor meu dizia[1], "você não precisa gostar da convenção, você precisa apenas segui-la".

Sobre a escolha do nome, seguem algumas outras regras importantes (MARTIN, 2008):

- Use nomes pronunciáveis; evite siglas e abreviações. Ao invés de "codFunc", prefira "codigoDoFuncionario".
- Evite codificações e prefixos. Ao invés de "b_antigo" ou "boolAntigo", prefira "éAntigo".

[1] O professor Paulo Feofiloff, autor do livro "Algoritmos em linguagem C" (FEOFILOFF, 2008).

- Use nomes do domínio do problema. Essa é uma regra essencial da Orientação a Objetos, já que a OO é uma forma de abstração do domínio do problema (como veremos). Se no seu problema o identificador de um produto é chamado de "código", para referenciá-lo no programa use "código" e não "identificador".
- Seja consistente: use a mesma palavra para um determinado conceito. Não use palavras diferentes para denotar um mesmo conceito. Por exemplo, se "adicionar", "inserir" e "guardar" significam a mesma coisa, então use somente um desses nomes (por exemplo, "adicionar").
- Evite nomes exageradamente longos. Por mais que bons IDEs facilitem a escrita de código, é desagradável ler uma variável chamada "nomeDasPessoasQueAindaNaoEstaoCadastradas". Prefira "nomesNaoCadastrados".

Formatação

O espaçamento e as quebras de linha são importantes para qualquer texto. Por exemplo, leia o fragmento do livro *Dom Casmurro* na Figura 1. Ele segue todas as regras da língua portuguesa. O único detalhe é que ele tem uma formatação *diferente*.

Traziam não sei que fluido
misterioso e enérgico ,uma força que arrastava para dentro , como a vaga que se retira da praia, nos dias de ressaca.Para não ser arrastado,agarrei- me às outras partes vizinhas,às orelhas,aos braços,aos cabelos espalhados pelos ombros, mas tão

depressa buscava as pupilas, a onda que saía delas vinha crescendo, cava e escura , ameaçando envolver-me, puxar -me e tragar - me.Quantos minutos gastamos naquele jogo
?Só os relógios do céu terão marcado esse tempo infinito e breve .

Figura 1: Fragmento do livro Dom Casmurro (ASSIS, 1994, p. 64), com problemas de formatação.

Mesmo um texto como esse se torna difícil de ler por causa de alguns espaços e quebras de linha em lugares *estranhos*. Isso não é diferente para um código fonte! Ainda que você seja um "Machado de Assis" da programação, o seu código será difícil de ler por um outro programador se ele não for formatado adequadamente.

Uma parte importante da formatação é a *indentação*. A indentação é o espaçamento colocado no começo de uma linha para identificar um *bloco*. Comandos que estão no mesmo nível (ou seja, no mesmo bloco), deverão ter o mesmo espaçamento no começo da linha. Blocos aninhados (ou seja, internos a um outro bloco) deverão ter um espaçamento maior que o do bloco externo. Isso é usado para identificar rapidamente o aninhamento de comandos em blocos. Por exemplo, considere o seguinte código:

```
1  if (quantidade > 0) {
2    total += quantidade;
3    if (total >= MAXIMO)
4      total = MAXIMO;
5  }
```

Por mais que existam chaves para marcar o início e o fim do bloco, o espaçamento deixa mais evidente que os comandos nas linhas 2 e 3 estão dentro do bloco do `if` da linha 1. O código da linha 4, por mais que não esteja dentro de chaves, só será executado se a condição do `if` da linha 3 for verdadeira – e isso fica claro com a indentação.

Uma outra parte da formatação que é muitas vezes menosprezada é o espaçamento, seja ele entre *palavras* ou entre linhas. Por exemplo, na linha 4 do código acima há 1 espaço entre "`total`", "`>=`" e "`MAXIMO`", mas

não há espaço entre "`;`" e "`MAXIMO`". Esse espaçamento, apesar de parecer arbitrário, segue a regra de espaçamento usada em textos em português. A sugestão é seguir essas mesmas regras ao programar. De uma forma geral, nomes de variáveis, tipos, comandos (`if`, `while`, `for`, etc.), e símbolos de comparação e atribuição ("=", "==", "<", "<=", etc.) são considerados *palavras*. Os símbolos "`;`" e "`,`" são considerados sinais de pontuação e eles e os parênteses seguem as mesmas regras de formatação de textos em português. Note que "`.`", chaves e colchetes *não são* considerados sinais de pontuação e, portanto, seguem regras específicas.

O espaçamento entre linhas também é importante para a legibilidade. O excesso de linhas em branco torna o código maior do que deveria e também cria uma quebra desnecessária na leitura. Por outro lado, a falta de linhas em branco torna o código mais denso e mais difícil de ler. De uma forma geral, evite deixar mais de uma linha em branco – duas ou mais linhas em branco é um espaçamento desnecessariamente grande. Use uma linha em branco para criar uma *seção* no seu código: coloque-a quando você desejar separar um conjunto de ideias de outro. É normal colocar uma linha em branco entre funções, entre métodos, ou entre atributos e métodos. Mas você também pode usar uma linha em branco dentro de uma função ou método para criar uma separação "lógica" no seu código. Por exemplo, se para fazer uma transferência entre uma conta corrente e outra é necessário retirar da conta e depois depositar na conta destino, então você pode separar o conjunto de comandos relativos à retirada dos de depósito com uma linha.

A seguir são apresentadas algumas regras de indentação e de espaçamento em geral, baseadas no livro do prof. Paulo Feofiloff (2008):

1. Acrescente uma tabulação na indentação ao iniciar um bloco e retire-a ao terminar o bloco;
    - Caso um bloco tenha apenas uma linha, pode-se deixa-lo na mesma linha do comando (`if`, `while`, `for`, etc.);
2. Use um espaço para separar duas palavras;
3. Deixe um espaço depois, mas não antes de um sinal de pontuação ("`;`" e "`,`").
4. Deixe um espaço antes, mas não depois de abrir um parêntese ("`(`");
5. Deixe um espaço depois, mas não antes de fechar um parêntese ("`)`");
6. Deixe um espaço antes e pule uma linha depois de abrir uma chave ("`{`");
7. Pule uma linha antes e depois de fechar uma chave ("`}`");
8. O fechamento da chave ("`}`") deve estar tabulado da mesma forma que o comando que abre a chave ("`}`");
9. Não coloque espaço antes e depois de abrir um colchete ("`[`");
10. Não coloque espaço antes, mas coloque depois de fechar um colchete ("`]`"); e
11. Não coloque espaço nem antes e nem depois de um ponto final ("`.`").
12. Pule uma linha entre funções e entre métodos.
13. Limite o tamanho da linha a 80 caracteres.
    - Caso uma linha precise ter mais de 80 caracteres, quebre-a ao final de um comando e alinhe-a com duas tabulações (para diferenciar da abertura de um escopo).

Outras sugestões de organização geral:

1. Declare variáveis no início do bloco (*lógico* ou *real*);
    - O bloco lógico é uma parte do código que faz algo específico, mas que não é necessariamente um bloco real (entre abre e fecha chaves); e
2. Declare variáveis de controle de `for` na primeira parte do `for`.

Na Tabela 1 são apresentados alguns exemplos de formas corretas e erradas de se formatar, seguindo a convenção apresentada.

Tabela 1: Exemplos de formas corretas e erradas de se formatar.

Correto	**Errado**

`if (x < y)`	`if ( x< y )`
	`if ( x < y )`
	`if(x<y)`

`for (int i = 0; i < 5; i++)`	`for(int i=0;i<5;i++)`
	`int i; // se o i só for usado no for`
	`for (i=0; i<5; i++)`

`int algo, outro, k;`	`int algo,outro,k;`
	`int algo,`
	`  outro,`
	`  k;`

Note que não existe uma convenção rígida para o espaçamento ou mesmo para a indentação. Um bom código deve ser acima de tudo consistente.

Comentários

Uma dúvida comum de programadores iniciantes é quanto comentário um código deve ter. Já vi programadores orgulhosos em mostrar códigos Java com um comentário para cada linha de código. Que mal haveria nisso? Se o comentário serve para explicar o código, quanto mais comentários, melhor, certo?

Errado. Idealmente quanto menos comentários, melhor. Pode parecer um contrassenso, mas *um bom código não precisa de comentários*. Se um código é simples e legível, qualquer programador consegue entende-lo facilmente. Então se os comentários não *acrescentarem nenhuma informação*, eles serão redundantes e, além disso, desviarão a atenção do principal – que é o código. Um outro motivo importante para evitar comentários é a manutenção. Se o código for alterado, o comentário associado deve ser também atualizado para mantê-lo consistente, e isso custa tempo de desenvolvimento. Nada mais confuso do que encontrar um código que incrementa uma variável e ver o comentário falando que a variável precisou ser decrementada!

Um bom programador deve se esforçar ao máximo para escrever códigos legíveis que não precisam de comentários. Um comentário não deve ser uma forma de compensar pelo código ser ruim (MARTIN, 2008)! Se o código é confuso, tente reescreve-lo para torna-lo legível.

Então para que servem comentários? Um bom código *nunca* terá comentários? Não é bem assim. Algumas informações não conseguem ser expressas no código, mas precisam ser passadas para quem está o lendo. Essas informações podem ser avisos em geral (por exemplo, que um algoritmo toma muito tempo), avisos de que o código está incompleto (colocando um comentário que há algo "a fazer" – representados normalmente como um TODO, do inglês "*to do*"), explicações do motivo do código ser assim (por exemplo, que um código varre um arranjo de trás para frente porque normalmente o valor procurado está no final do arranjo), entre outras. Portanto, só escreva um comentário se ele acrescentar algo que não pode ser expresso na linguagem de programação.

Por fim, comente um fragmento de código apenas se não for necessário usá-lo *temporariamente*. Se o código não tem previsão de ser usado novamente, apague-o. Ter um código com várias linhas comentadas atrapalha a leitura e naturalmente gera a dúvida do porquê aquilo estar comentado ("será que isso é importante?"). Existem maneiras melhores de manter o histórico do código e que não atrapalham a legibilidade: os sistemas de controle de versão. Por exemplo, o Git (https://git-scm.com), o Mercurial (https://www.mercurial-scm.org) e o CVS (http://www.nongnu.org/cvs/).

Otimização

Uma das qualidades de um bom código é ele ser *eficiente*. Mas o que fazer se a eficiência afetar a elegância? Um exemplo simples é encontrar o menor número em dois arranjos não vazios. O que será que é melhor, (1) criar uma *função* que encontra o menor valor em dois arranjos ou (2) criar uma *função* que encontra o menor em um arranjo, chamando-o duas vezes? Pensando em eficiência, a opção (1) é definitivamente melhor: chamadas de métodos são custosas e, além disso, se aproveitaria variáveis - note que na opção (2) em cada chamada de método é preciso criar uma variável "menor", além de se guardar o "menorDoArranjo1" e o "menorDoArranjo2". Mas a opção (2) é mais elegante: o código é muito mais simples de entender. Assim como nesse exemplo, muitas vezes é necessário diminuir a legibilidade de um código para torna-lo eficiente. Então o que fazer? Quem devemos privilegiar, a *elegância* ou a *eficiência*?

Bem, depende! Em alguns sistemas embarcados ou críticos a eficiência pode ser fundamental. Uma variável a mais pode fazer com que o código não caiba na memória disponível; uma chamada de método pode consumir alguns ciclos valiosos do processador. Nesses casos, a legibilidade perderá para a eficiência. Mas na *grande maioria* dos cenários atuais o computador tem capacidade de processamento e memória suficiente para arcar com a perda de desempenho causada por tornar um código elegante. As alterações necessárias têm um impacto muito pequeno na eficiência: no máximo serão criadas algumas variáveis a mais ou serão feitas chamadas de métodos a mais. Pensando em análise de algoritmos, isso não mudará a complexidade do algoritmo: se ele for quadrático ele continuará sendo quadrático[2]. Mas essa variável a mais pode fazer com que um código demore alguns minutos a menos para ser entendido por um outro desenvolvedor, ou mesmo evitar que uma alteração leve a um defeito.

Portanto, a menos que a eficiência seja crucial (um sistema embarcado, crítico ou uma parte crítica do código), privilegie a elegância. O ganho de desempenho será desprezível frente aos problemas que ele pode gerar. Citando Donald Knuth, um dos grandes nomes da computação e o pai da análise de algoritmos (1974, p. 671):

> *O problema real é que programadores gastaram tempo demais se preocupando com eficiência nos lugares errados e nas horas erradas; otimização prematura é a raiz de todo mal (ou pelo menos boa parte dele) na programação.*

Se isso era verdade ao considerar o poder de processamento de computadores da década de 1970, isso é ainda mais verdade atualmente. Mas isso não quer dizer que não devemos nos preocupar em criar códigos eficientes. Um bom programador deve sempre tentar privilegiar tanto a elegância e a eficiência. Então se uma parte do código está confusa, tente reorganizá-la para torna-la mais fácil de entender. Altere os nomes das variáveis e dos métodos (isso não afeta a eficiência); se ajudar, crie métodos e variáveis auxiliares.

[2] Veja o livro do (CORMEN et al., 2012) sobre análise de algoritmos.

Duplicação de código

Apesar de parecer inofensivo, a duplicação de código gera problemas sérios de manutenção. Imagine que um mesmo código está em 4 lugares diferentes de um projeto. Um desenvolvedor então descobre que há um defeito grave nesse código e decide corrigi-lo. Se ele não souber que o código está duplicado, ele só alterará uma das cópias do código – o resto ficará com o defeito! O mesmo acontece se ao invés de um defeito for uma melhoria: ela não será propagada automaticamente. Isso sem falar que a duplicação aumenta desnecessariamente a base de código (além do código duplicado, os testes também devem ser duplicados, por exemplo).

Portanto não duplique código. Se for necessário usar um código igual ou similar a um que já existe em algum lugar, refatore-o para que seja possível reaproveita-lo. Às vezes pode ser necessário generalizar alguma parte do código (por exemplo, adicionar um parâmetro a mais no método) para conseguir reaproveita-lo adequadamente. E tenha sempre em mente os conceitos da Orientação a Objetos. Um dos benefícios da OO é exatamente permitir o reuso!

Referências

ASSIS, M. DE. **Dom Casmurro**. 3. ed. [s.l.] FTD, 1994.

CORMEN, T. H. et al. **Algoritmos. Teoria e Prática**. Edição: 3ª ed. Rio de Janeiro: Elsevier, 2012.

FEOFILOFF, P. **Algoritmos em linguagem C**. 1. ed. [s.l.] Elsevier, 2008.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. 1 edition ed. Reading, MA: Addison-Wesley Professional, 1999.

GRUBB, P.; TAKANG, A. A. **Software Maintenance: Concepts and Practice**. 2 edition ed. River Edge, N.J: World Scientific Pub Co Inc, 2003.

KNUTH, D. E. Computer Programming As an Art. **Commun. ACM**, v. 17, n. 12, p. 667–673, dez. 1974.

MARTIN, R. C. **Clean Code: A Handbook of Agile Software Craftsmanship**. 1. ed. Upper Saddle River, NJ: Prentice Hall, 2008.

PFLEEGER, S. L.; ATLEE, J. M. **Software Engineering: Theory and Practice**. 4 edition ed. Upper Saddle River N.J.: Pearson, 2009.

SUN MICROSYSTEMS. **Java Code Conventions.** Sun Microsystems, 1997. Disponível em: <http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>

```c
#define _GNU_SOURCE // necessrio porque getline()  extenso GNU
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct jogador {
    int nivel;
    char nome[30];
} jogadores[50], aux;

int j = 0;

void bubble_sort()
{
    for (int i = 0; i <j; i++)
    {
        for (int j = 0;j < j -1; j++)
        {
            if (jogadores[j].nivel > jogadores[j+1].nivel){
                aux = jogadores[j];
                jogadores[j] = jogadores[j+1];
                jogadores[j+1] = aux;}
        }
    }
}


void carregaJogadores()

{
  FILE *pont_arq;
  size_t len= 100;
  char *linha =malloc(len);

  pont_arq = fopen("arquivo.txt", "r") ;


  char delim[] = ",";

  while (getline(&linha, &len, pont_arq) > 0) {
    char *ptr = strtok(linha, delim);
    if ((ptr != NULL) && (!strcmp(ptr,"1")) )

    {
        ptr = strtok(NULL, delim);
        jogadores[j].nivel = atoi(ptr);
        ptr = strtok(NULL, delim);
        strcpy(jogadores[j].nome, ptr);
        j++;
    }
  }
  if (linha)
    free(linha);
```

```
54
55    bubble_sort();
56
57    for (int i=0; i<j; i++)
58    {
59        printf("%d) %s",jogadores[i].nivel,jogadores[i].nome);
60    }
61
62    fclose(pont_arq);
63  }
64
65
66  void teams(int nTimes)
67  {
68      char vTimes[nTimes][300];
69      for (int i = 0; i<nTimes; i++)
70      {
71          strcpy(vTimes[i],jogadores[i].nome);
72      }
73
74      for (int i = nTimes; i < j; i++)
75      {
76          strcat(vTimes[(nTimes-1)-(i%nTimes)],jogadores[i].nome);
77      }
78
79
80
81      printf("\nTimes Selecionados:\n\n");
82      for (int i=0; i<nTimes; i++)
83      {
84          printf("Time %d)\n%s\n",i+1,vTimes[i]);
85      }
86  }
87
88
89
90  int main(void){
91      int n;
92    carregaJogadores();
93    printf("Quantos Times? ");
94    scanf("%d",&n);
95    teams(n);
96    return(0);}
```