Universidade de Brasília

Departamento de Ciência da Computação

Disciplina: Métodos de Programação – 201600

Aluno: Aécio Fernandes Galiza Magalhães – 15/0115121

Trabalho 2 – Requisitos para o desenvolvimento de Software

I. Como fazer levantamento de Requisitos

A seguir, serão listados os itens necessários para verificação da qualidade do levantamento de requisitos:

1) **Os requisitos levantados estão escritos em um nível de detalhes apropriado e consistente?** – A fim de se validar com clareza (na relação entre desenvolvedor e cliente) o que realmente está sendo construído como produto. Por exemplo: os requisitos "um sistema capaz de realizar transações entre contas bancárias" e "um sistema capaz de realizar transições entre contas bancárias do mesmo Banco, com valores não maiores do que 5 mil reais" possuem níveis de granularidade diferentes. O detalhamento deve ser suficiente para o desenvolvimento de um programa que supra as necessidades do cliente.
2) **As ferramentas (interfaces de hardware, software, comunicação) estão todas definidas?** – A importância desse requisito se dá pela necessidade de uma boa relação de comunicação entre o produto e possíveis interfaces adicionais.
    - **i.** As **interfaces de software** devem conter todos os componentes adicionais necessários para o funcionamento do produto, incluindo nome e versão necessárias, bem como a descrição da necessidade desses conteúdos externos e como/quando serão utilizados;
    - **ii.** As **interfaces de hardware** devem ser descritas como equipamentos suportados, como é feita a interação hardware-software e os protocolos utilizados para essa comunicação;
    - **iii.** As **interfaces de comunicação** devem ser bem definidas, como a necessidade de uso de serviços de web, protocolos de network, bem como especificar segurança, como funciona a transferência de dados e como os serviços são sincronizados.
3) **O comportamento do sistema frente à antecipação de erros está bem documentado?** – Deve ser descrito as condições para os erros previstos que podem ocorrer durante o caso do usuário, e como o sistema deve responder à esse erro, assim como no caso de falhas de não-execução, bem como o comportamento que deve ser tomado frente à estas. Por exemplo: na tentativa de inserção de uma quantia negativa para transferência bancária, será lançada uma exceção, com descrição do porquê do erro ter ocorrido (valor negativo) e o que deve ser feito (inserir uma quantia válida).
4) **Algum dos requisitos está duplicado ou vai de encontro a outro requisito já descrito?** – Dada a boa definição de um requisito no que se refere ao detalhamento e consistência (1), este deve ser completo e único, não havendo um outro requisito que o rediga ou o contrarie. Por exemplo: o requisito "um

sistema capaz de realizar transições entre contas bancárias do mesmo Banco, com valores não maiores do que 5 mil reais" e a descrição "poderão ser realizadas transferências entre valores de 20 reais a 10 mil reais" mostram-se contraditórios.

5) **O requisito está claro, conciso e não-ambíguo?** – O requisito, por si só, deve ser completo, evitando a necessidade de maiores explicações acerca deste. Um requisito ambíguo gera desentendimento no escopo do produto, podendo acarretar problemas na relação entre o desenvolvedor e o cliente. Por exemplo: ao alegar que "o sistema deve gerar um resultado consideravelmente rápido", temos um problema de clareza, ao se desconhecer o que seria "consideravelmente rápido". Quais são os parâmetros? Rápido com relação a que? A definição "o sistema deve gerar um resultado em um tempo menor que 15 segundos" define a imagem das possibilidades de tempo de resposta.
6) **Os requisitos estão livres de erros gramaticais? –** Para a documentação dos requisitos, estes devem seguir uma linguagem formal e livre de erros.
7) **As questões de segurança estão bem especificadas? –** No exemplo dado anteriormente, tratamos com algumas questões que necessitam de segurança dos dados. Assim, deve ser garantido que as informações pessoais (por exemplo, de cadastro, transações) não são divulgadas e que informações críticas (senhas) são criptografadas, impedindo uma vulnerabilidade interna.
8) **Cada requisito é único e está bem classificado?** – A necessidade se dá pela possibilidade de ligação de uma necessidade com um requisito; verificar as aplicações do requisito e garantir sua unicidade. Por exemplo, através de um sistema lógico de classificação de especificações e requisitos, pode ser montada uma tabela de relações entre estes, provando sua aplicação.
9) **Todos os requisitos definidos são realmente requisitos?** – Os requisitos são definidos através da descoberta das necessidades dos **stakeholders**, documentando elas para futuras consultas e, posteriormente, implementação. Não devem haver soluções de design ou implementação no escopo dos requisitos. Por exemplo: não permitir requisitos definidos como implementações, como no caso de "através da autenticação do cliente, deve-se acessar sua conta no banco de dados, retirar uma quantia de sua conta e adicionar essa quantia na conta futura da transferência", já que este já faz parte de como será feita a implementação.
10) **Todos os requisitos instáveis estão especificamente classificados como tal? –** Os requisitos instáveis devem possuir a *flag* "TBC" – To be confirmed – ou equivalente; este requisito pode depender constantemente de *feedbacks* e interações com o cliente, estando sujeito a mudanças futuras.

II. Como fazer o *Design* do Software

1) **A nomenclatura das variáveis segue um padrão?** – Uma vez definido o padrão, este deve ser seguido durante o desenvolvimento do Software, por exemplo, uma variável definida no padrão **lowerCamelCase** é declarada como "nomeMes".
2) **A nomenclatura das variáveis representa seu significado?** – Ao se deparar com uma variável "mês", a princípio, não podemos definir com certeza se esta se trata do nome do mês, ou seu representativo em numeral. Assim, optar por definir uma especificação a mais para o nome da variável, por exemplo, "**nomeMes"**, evitando a necessidade de acrescentar linhas de comentário supérfluas, ou de se verificar novamente em outra parte do código o que ela deveria representar.

3) **Nomenclatura de variáveis como constantes –** Para a padronização, optar por declara variáveis constantes em letra maiúscula, com as palavras separadas pelo caractere "_". Por exemplo: "**const unsigned TOTAL_MESES = 12;**".
4) **Nomenclatura de classes** – O nome de uma classe deve ser um substantivo, declarado com a letra inicial de cada palavra que a compõe em maiúsculo. Por exemplo, em se referindo ao projeto do software de transações, podemos ter a classe "ContaCorrente".
5) **Classes com padrões de projeto –** Classes responsáveis por algum *design pattern* devem ser declaradas com um sufixo que represente esse padrão. Por exemplo, na classe "Usuario**Facade**", onde temos o design pattern "Facade" para a classe de usuário.
6) **Nomenclatura de métodos de Ação Direta** – Para métodos que executam uma única simples ação, temos definições de nomenclatura básica, como por exemplo "create()", "init()", "clear()", entre outros.
7) **Nomenclatura de métodos de Ações Simples** – é executada uma ação simples, a qual necessita de um objeto, que deve ser descrito no nome do método. Por exemplo "sendMessage()", "beginTransaction()", "showFirstResult", entre outros.
8) **Nomenclatura de métodos get/set** – Para se obter as propriedades de uma classe da forma correta, devem-se utilizar os métodos getters/setters, na forma "getAccountNumber", "setTransactionDate", "getTransactionDate", entre outros.
9) **Nomenclatura de métodos booleanos** – Ao se recuperar um resultado booleano, declaramos o método na forma de pergunta, como por exemplo "isEmpty", "isFull", "isAvaliable", "hasAccount", entre outros.
10) **Ordem de inclusão de headers** – Os headers devem ser incluídos na seguinte ordem: Header relacionadio > header do sistema C > header de biblioteca padrão C/C++ > header de outras bibliotecas > header do seu próprio projeto, separando-se cada grupo não-nulo com uma linha em branco.
11) **Uma função deve performar uma operação simples** – Com esse requisito, é possível corrigir, testar, entender e reutilizar uma função de uma forma mais fácil.
12) **Se uma função é pequena e crítica, ela está declarada como inline**? **–** Se uma função é resumidamente simples, optar por declarar na forma **inline**, a fim de se utilizar algumas otimizações do compilador. Por exemplo: "inline int Total(int OperandoUm, int OperandoDois) {return OperandoUm + OperandoDois;}"
13) **Preferência por bibliotecas padrão ou outras bibliotecas do que recriar um código** – Para funções muito bem conhecidas, evitar recriar o código para elas, quando se pode utilizar uma biblioteca padrão pronta ou outra biblioteca.
14) **Declarar variáveis em escopo local limitado sempre que possível** – Por exemplo, ao se utilizar de um contador apenas para um **for**, é interessante declarar como "for (int i = 0; i < 10; i++)". Assim, favorece a legibilidade do código.
15) **Manter uma descrição de nome de variável curta para variáveis simples e locais e longa para variáveis não-locais –** Também para melhorar a legibilidade, declarar de forma mais descritiva variáveis de escopo não-locais, e de forma resumida variáveis locais e menos relevantes. Por exemplo, para um contador, a variável "int i" é ideal. Contudo, essa variável representaria de uma péssima forma um valor para, por exemplo, o saldo de uma conta bancária.

16) **Evitar nomes muito parecidos para variáveis** – Também na questão de legibilidade de código, evitar nomes muito parecidos, como, por exemplo, “ol, oI, o1”, entre outros.
17) **Opte por declarar apenas um nome por declaração de variável** – Para melhor legibilidade e para evitar possíveis problemas de sintaxe da gramática em C/C++, **evitar** declarações do tipo int *a, b, c[1], *d[1], **e[1]; Optar pela declaração unitária **em cada linha**.
18) **Não reutilizar nome de variável em escopos aninhados** – É fácil se confundir ao utilizar o mesmo nome de variável em escopos aninhados, mesmo que não haja erro sintático ou semântico no código. Opte pela criação de novas variáveis quando se tratando do mesmo escopo.
19) **Sempre inicialize a variável em sua atribuição** – A fim de se evitar erros ao longo do código, reforçando o requisito da seção [III.5], **sempre** inicialize a variável em sua declaração, por exemplo, “int numero = 10”.
20) **Preferência pela utilização de “nullptr” ao invés de 0/NULL** – Para legibilidade, **nullptr** não pode ser confundido com um inteiro. Além disso, é uma declaração mais geral, que pode funcionar em mais cenários.

III. Como fazer a codificação

1) **Os Arrays declarados estão dimensionados para lidar com constantes? –** Alguns projetos exigem parâmetros constantes para executar suas ações em se tratando de um âmbito fixo. Por exemplo, em se tratando de uma tarefa que envolva a criação de um vetor incluindo todos os meses do ano. Assim, temos, por definição, que o ano possui 12 meses. Ao invés de se criar um vetor **meses[13]**, decide-se por criar um vetor **meses[TOTAL_MESES + 1]**, sendo total uma constante igual a 12.
2) **As variáveis imutáveis estão declaradas como constante?** – As variáveis que não podem estar sujeitas a mudanças ao longo do código **devem** ser declaradas como **constantes**. Assim, a variável antes mencionada “total_meses” deve ser definida como “**const unsigned TOTAL_MESES = 12;**”.
3) **Constantes não devem ser declaradas como #define, e sim como const** – Alguns códigos trazem constantes declaradas com o macro “#define”. O problema dessa abordagem é que essa declaração simplesmente serve para substituir o texto no local onde ele é encontrado, e muitas vezes acaba gerando problemas de interpretação, principalmente em se tratando de constantes para inteiros. Assim, ao invés de declarar “#define TOTAL_MESES 12”, opte por “**const unsigned TOTAL_MESES = 12;**”.
4) **Valores negativos para a variável declarada fazem sentido**? **–** Caso a resposta seja “não”, esta deve ser declarada como **unsigned**. Por exemplo, na questão do total de meses do ano, “**const unsigned total_meses = 12;**”.
5) **As variáveis estão declaradas antes de seu uso?** – As variáveis de uso local devem ser sempre declaradas antes de seu uso, a fim de se evitar que, por ventura, um código passe pela fase de compilação por possuir uma variável global usada em escopo local (erroneamente), quando na verdade o desenvolvedor deveria ter utilizado uma variável local. Assim, não é alegado nenhum erro de compilação, contudo o resultado não é o esperado.
6) **As variáveis são declaradas em uma parte do código e posteriormente atribuídas? –** A declaração em si de uma variável atribui um valor (mesmo se não especificado) a ela. Essa “dupla atribuição”, na declaração e posteriormente na atribuição, gera um **overhead**. Assim, as variáveis devem ser declaradas e já atribuídas com o valor que deverá possuir na sua utilização.

7) **O argumento de qual quer-se extrair o "sizeof" é o correto?** – Alguns erros mostram-se comuns, como, por exemplo, ao querer extrair o tamanho de um elemento de um array, deve ser feito sizeof(array[elemento]), e **não** sizeof(array).
8) **Os arrays estão sendo destruídos da forma correta?** – A forma para se destruir um **array** deve considerar sua magnitude, logo, deve ser feita como "**delete [] meses"**.
9) **Os elementos deletados terão ponteiros apontando para estes?** – É ideal que um elemento deletado possua alguma forma de identificar que foi realmente deletado, a fim de se facilitar a interpretação do próximo requisito descrito. Assim, é recomendado que os ponteiros estejam apontando para **NULL** após a destruição.
10) **O código está tentando destruir elementos que já foram destruídos?** – Visto a implementação seguindo o requisito acima, não há a necessidade de confirmação deste, pois é sempre seguro deletar ponteiros que já apontam para **NULL**.

Referências

[1] Davis, A.; Overmyer, S.; Jordan, K.; Caruso, J.; Dandashi, F.; Dinh, A.; Kincaid, G.; Ledeboer, G.; Reynolds, P.; Sitaram, P.; Ta, A.; Theofanos, M.: Identifying and measuring quality in a software requirements specification. In: Proc. 1st International Software Metrics Symposium, pp. 141-152. (1993).

[2] Wilson, W.M.; Rosenberg, L.H.; Hyatt, L.E.: Automated analysis of requirement specifications. Proceedings of the 19th International Conference on Software Engineering. (1997).

[3] Swathi, G.; Jagan, A.; Prasad, Ch: Writing Software Requirements Specification Quality Requirements: An Approach to Manage Requirements Volatility. Int. J. Comp. Tech. Appl., 2(3), 631-638. (2011).

[4] Sommerville, I.: Engenharia de Software. Tradução Ivan Bosnic e Kalinka G. de O. Gonçalves . Revisão Técnica Kechi Hirama. 9. ed. São Paulo : Pearson Prentice Hall, 2011.