Fundação CECIERJ - **Vice Presidência de Educação Superior a Distância**

Curso de Tecnologia em Sistemas de Computação
Disciplina: Engenharia de Software
AP2 1° semestre de 2019.

1) O que é Divida Técnica? Qual sua relação com a qualidade do software? Como pode ser observada? (valor 2,0 pontos)

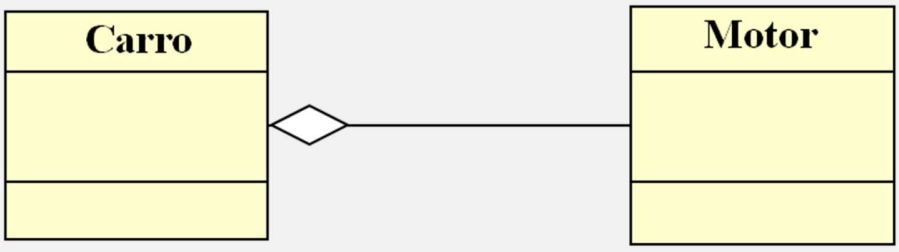
VER ANEXO (ULTIMA PAGINA)

2) Explique o que é um módulo em um projeto de desenvolvimento de software e relacione a importância de um módulo com o princípio da separação de objetivos. (valor 2,0 pontos)

Um módulo é um componente bem definido de um sistema; uma unidade que pode ser compilada de forma independente das demais unidades. Módulos podem conter rotinas, dados e definições de tipos. Em uma relação de uso, um módulo pode ser visto como cliente e o outro módulo como fornecedor de serviços. O comportamento do módulo cliente deve ser compreensível sem a análise detalhada de seus fornecedores.

3) O que representa uma relação de composição em um diagrama de classes UML? Quais são os relacionamentos entre classes/objetos que podem ser representados neste diagrama? Apresente um exemplo de diagrama de classes e objetos. (valor 2,0 pontos)

Uma composição representa uma relação onde há dois papéis: *container* e conteúdo, como em um cenário em que um carro contém um motor. Os relacionamentos entre classes são a herança, a composição e as associações. Podemos ainda mencionar os auto-relacionamentos, que são um tipo especial de composição ou associação. Para o último item da pergunta, qualquer diagrama de classes que apresente ao menos duas classes e um relacionamento entre elas é válido (como no exemplo abaixo).



4) Que percepção a medida da Complexidade Ciclomática pode fornecer sobre o software? Extraia a medida do valor da métrica (número ciclomático) para o grafo de fluxo de controle apresentado na Figura 1 (derivado do programa apresentado). O que este valor (ou a variação dele) pode ajudar no planejamento dos testes e atividades de manutenção? (valor 2,0 pontos)

Esta métrica fornece uma medida quantitativa da complexidade lógica de um programa. No contexto do teste estrutural, seu valor define o número de caminhos independentes e nos fornece o número máximo de casos de teste que garantem que todos os comandos tenham sido executados pelo menos uma vez, o que fornece uma expectativa de esforço de teste e manutenção.

Seu cálculo se dá através da identificação do número de regiões no grafo de fluxo do programa, através da fórmula:

V(G)=E-N+2
E: número de arcos
N: número de nós
ou,

V(G) = P +1
P: número de nós predicados (decisões)

Neste caso, o valor da Métrica é 6 (17 arcos, 13 Nós) ou (5 Nós Predicados: 2, 3, 5, 6 e 10)

5) Explique o que é teste estrutural. Diferentes critérios podem ser utilizados para projetar casos de teste. Projete casos de teste para o código apresentados na Figura 1 utilizando o critério todos os nós e o critério todas as arestas. Qual deles apresenta maior cobertura de teste? Por quê? (Modelos extraídos do material preparado pelo Prof. Andrey Ricardo Pimentel, UFPR, www.inf.ufpr.br/andrey/ci221/apresentacaoTesteEstrutural.pdf , *23/04/2019)* (valor: 2,0 pontos)

Teste estrutural, ou teste caixa aberta, utiliza a informação sobre a estrutura do software para o planejamento dos testes. Neste sentido, a organização utilizada para a construção do software pode ser servir para apoiar os diferentes critérios de teste. Por exemplo, o teste pode ser inspirado no fluxo de controle do software ou no fluxo de dados. Assim, podemos usar como critérios percorrer todos os nós, arcos, caminhos, uso de dados, e demais possibilidades.

No modelo apresentado, a aplicação do critério todos os nós (todas as instruções) implica em criar caminhos, cada qual correspondente a um caso de teste, no qual cada caso de teste deve percorrer pelo menos um nó ainda não percorrido anteriormente até que todos os nós tenham sido executados.
Para isso,
Caminho 1: Nós {1,2,3,4,5,6,7,8,9,2,10,12,13}
Caminho 2: Nó {2 (desvio para o nó 10), 10 (desvio para o nó 11), 11}
Caminho 3: Nó {3 (desvio para o nó 10)}
Caminho 4: Nó {5 (desvio para o nó 8)}
Caminho 5: Nó {6 (desvio para o nó 8)}

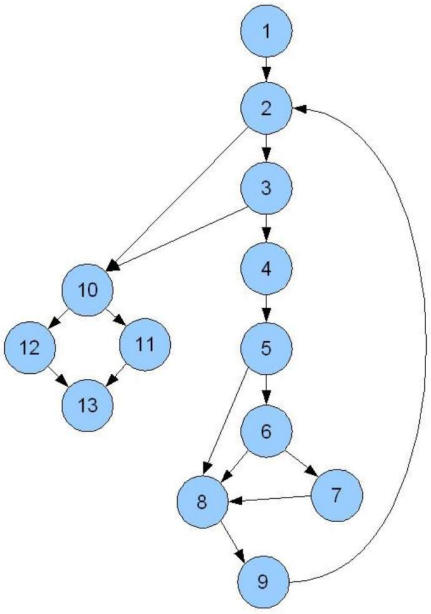
Para percorrer os nós, os valores de entrada de Valor[i], min e Max devem ser ajustados para satisfazer as condições de desvio do código:
Caminho 1: Valor[i] <> -999, min >= Valor[i]; Max <= Valor[i]
Caminho 2: Valor [i] = -999, validas < 0
Caminho 3: entradas = 100
Caminho 4: Valor[i] < min
Caminho 5: Valor[i] > max

No modelo apresentado, a aplicação do critério todas as arestas (todos os caminhos) implica em percorrer uma aresta (caminho) pelo menos uma vez no conjunto de todos os casos de teste, rotulam-se os caminhos e criam-se os casos de teste para percorrer cada caminho, até que todos eles tenham sido percorridos. Para isso, seis casos de teste são necessários para testar a aplicação:
Caminho 1: Nós { 1, 2, 3, 4, 5, 6, 8, 9, 2, 10, 12, 13}
Caminho 2: Nós { 1, 2, 10, 12, 13}
Caminho 3: Nós { 1, 2, 3, 10, 11, 13}
Caminho 4: Nós { 1, 2, 3, 4, 5, 8, 9, 10, 11, 13 }
Caminho 5: Nós { 1, 2, 3, 4, 5, 6, 7, 8, 9, 2, 3, 11, 13}
Caminho 6: Nós { 1, 2, 10, 11, 13}

Para percorrer os caminhos, os valores de entrada de Valor[i], min e Max devem ser ajustados para satisfazer as condições de desvio do código:
Caminho 1: Valor[i] <> -999, min >= Valor[i]; Max <= Valor[i]
Caminho 2: Valor [1] = -999; min <= qualquer; Max >= qualquer
Caminho 3: Intangível. Como fazer entradas > 100?
Caminho 4: Valor[i] <> -999, min <= Valor[i]; Max >= Valor[i]
Caminho 5: Valor[i] <> -999, um Valor[i] <= min; Max <= Valor[i]

Caminho 6: Intangível. Como fazer entradas > 100 antes da entrada do Loop?

Do ponto de vista de cobertura, os nós são cobertos por ambos os critérios. Entretanto, o critério todas as arestas permite ter uma melhor cobertura de fluxo de controle, tendo em vista que os nós são percorridos na ordem com que deveriam ser executados e explorando a funcionalidade esperada do sistema.




```
Procedimento media
INTERFACE ACEITA valor, min, max
INTERFACE RETORNA media, entradas, validas

var
   valor[1..100] vetor de real
   media, entradas, validas, min, max, soma: real
   i : inteiro
inicio
   i = 1
1  totalEntradas = 0
   totalValidas = 0
   soma = 0                    2              3
   enquanto valor[i]<>-999 e entradas<100 faça
   4 entradas=entradas+1
                  5            6
      se valor[i]>=min e valor[i]<=max então
         validas=validas+1
   7     soma=soma+valor[i]
      senão pule
      fimse
   8  i=i+1
   9 fimenquanto
     se validas>0 então  10
   11   media=soma/validas
   12 senão
        media=-999
   13 fimse
fim
```

DÍVIDA TÉCNICA: DEFINIÇÃO E CONCEITOS

Este *evidence briefing* apresenta informações sobre a definição e os principais conceitos de dívida técnica.

CONCEITOS

O que é Dívida Técnica?
Dívida Técnica (DT) é um conceito cunhado inicialmente por Ward Cunningham em 1992, mas desde então tem recebido diversas atualizações. Ela foi adotada por profissionais de métodos ágeis, mas possui uma ampla aplicação em todos os domínios de software. A principal definição de DT é [1]:

"Em sistemas intensivos de software, dívida técnica é uma coleção de construtos de design ou de implementação que são efetivos no curto prazo, mas que criam um contexto técnica que torna mudanças futuras mais caras ou impossíveis. A dívida técnica apresenta um risco real ou de contingência, cujo impacto é limitado a qualidades internas do sistema, principalmente manutenibilidade e capacidade de evolução."

Quatro aspectos precisam ser observados:

- **A DT não incorre apenas em código fonte**. É possível identificar DT na elicitação de requisitos ou em casos de teste, por exemplo;
- Para ser considerado DT, o problema deve causar um **benefício de curto prazo, em troca de um potencial custo futuro**;
- DT só é associada a atributos de qualidade interna, como manutenibilidade, então **defeitos não são DT!** É possível existir, no entanto, um tipo de DT de defeito (veja mais a seguir);
- O correto é **dívida técnica** e não "débito técnico"! A metáfora está associada à ideia de um "empréstimo" feito pelo profissional, em troca de "pagamentos" futuros, que ficam mais caros quanto maior for o tempo para quitação da dívida.

Então, o que deve ser considerado DT [2][3]?
- Código mal escrito, que viole regras;
- "Atalhos" tomados durante o design;
- Defeitos conhecidos, cuja eliminação é adiada para *sprints* ou ciclos de desenvolvimento futuros;
- Problemas arquiteturais, como violação de modularidade;
- *Code smells*;
- Aspectos gerais de qualidade interna, que afetem a manutenibilidade e a capacidade de evolução.

E o que não deve ser considerado DT [3]
- Defeitos;
- Problemas triviais de qualidade de código, que não violem as regras de código;
- Falta de processos de suporte;
- Funcionalidades não implementadas.

Quais outros conceitos são associados com DT?
Como DT é uma metáfora financeira, relacionada com uma dívida adquirida por alguém para obter um ganho de curto prazo, alguns outros conceitos de finanças são frequentemente associados com DT, como:
- **Principal:** O esforço que é exigido para resolver a diferença entre o nível de qualidade atual e ótimo, em um artefato de software imaturo ou no sistema de software completo;
- **Juros:** O esforço adicional requerido para ser investido na manutenção do software, devido ao decaimento da sua qualidade;
- **Pagamento:** O esforço gasto na melhoria da qualidade do software. Este esforço diminuirá o esforço necessário para tarefas de manutenção futuras.

Como DT pode ser classificada?
A classificação mais simples de itens de DT pode ser pela sua intenção:
- **DT intencional** é aquela causada por decisões estratégicas e planejadas, quando o time ou a organização decidem obter um ganho de curto prazo ao custo de um esforço de longo prazo. Por exemplo, a decisão de desenvolver uma solução simplificada de arquitetura, sabendo que ela pode não atender as necessidades futuras do projeto;
- **DT não-intencional** não incorre com um propósito estratégico, e usualmente aparece em um projeto devido à imaturidade ou falta de conhecimento dos profissionais. Por exemplo, um código mal escrito criado por um programador inexperiente pode ser um item de DT.

Outra classificação possível de DT é pelo seu tipo, ou seja, pelo que causa aquele item de DT específico. Alves et al [2] reuniu 15 diferentes tipos de DT. Os mais recorrentes em projetos de software estão listados abaixo:
- **Dívida de design:** Associada principalmente com violações dos princípios de um bom design orientado a objetos, como classes muito acopladas;
- **Dívida de arquitetura:** Refere-se a problemas relacionados com a arquitetura do software, como violação de modularidade;
- **Dívida de documentação:** Dívida relacionada a problemas observados na documentação do software;
- **Dívida de teste:** Dívida encontrada em atividades de teste, como casos de teste planejados que não foram executados;
- **Dívida de código:** Associada com problemas encontrados no código-fonte, que podem tornar a manutenção mais difícil, usualmente relacionada com más práticas de programação;
- **Dívida de construção (build):** Refere-se a problemas que podem prejudicar a tarefa de build, gerando um consumo desnecessário de tempo.

Para quem é este briefing?

Profissionais de engenharia de software que quiserem tomar decisões sobre problemas de qualidade interna e aplicar conhecimento científico ao gerenciar a dívida técnica.

De onde vêm as informações?

As informações neste *briefing* vêm de evidência coletada pelo autor através de uma revisão na literatura em diversas publicações, incluindo:
- [1] Avgeriou, P. et al. Managing Technical Debt in Software Engineering. In Dagstuhl Reports, 2016;
- [2] Alves, N. S. et al. Identification and management of technical debt: A systematic mapping study. Information and Software Technology, 2016;
- [3] Li, Z. et al. A systematic mapping study on technical debt and its management. Journal of Systems and Software, 2015.

Para informações adicionais sobre o Grupo de Engenharia de Software Experimental na COPPE/UFRJ:

http://lens-ese.cos.ufrj.br/ese/

Para informações adicionais sobre o Observatório DELFOS:

http://www.delfos.cos.ufrj.br

REFERÊNCIA DE PESQUISA
Silva, V. M. ***Technical Debt and its Management: Analysis and Application in the Brazilian Software Industry***. Dissertação de mestrado, Universidade Federal do Rio de Janeiro, 2018.