UDESC
UNIVERSIDADE DO ESTADO DE SANTA CATARINA

PROCESSO SELETIVO – 02/2021

Área de Conhecimento: Engenharia de Software

PROVA ESCRITA – GABARITO

QUESTÃO 1: De acordo com Edsger W. Dijkstra, "Testes de software mostram a presença de bugs, mas não a sua ausência." Explique qual o seu entendimento em relação a esta frase. Ainda a respeito de testes de software, descreva porque testes podem ser considerados tanto uma atividade de verificação como de validação de software. Também especifique ao menos um tipo de teste adequado se o objetivo for verificação, bem como um tipo de teste adequado se o objetivo for validar um sistema de software.

RESPOSTA: Em relação a primeira parte da pergunta, é importante mencionar que a frase sintetiza não apenas os benefícios de testes, mas também suas limitações. Provar a ausência de bugs envolve executar o sistema com todos os possíveis dados de entrada, o que, na prática, é impossível. De acordo com PRESSMAN (2001), testes de software nunca terminam, a tarefa simplesmente passa do engenheiro de software para o cliente. Cada vez que o cliente/usuário executa um programa de computador, o software está sendo testado. SOMMERVILLE (2003) ainda afirma que os testes não podem demonstrar que o software é livre de defeitos para qualquer circunstância. Contudo, o teste é um processo que intenciona construir a confiança no software.

De acordo com PRESSMAN (2001) e SOMMERVILLE (2003), o teste é um elemento de um aspecto mais amplo, que é frequentemente referido como verificação e validação (V&V). Verificação tem como o objetivo garantir que um sistema atende à sua especificação. Já com validação, o objetivo é garantir que um sistema atende às necessidades de seus clientes. A diferença entre os conceitos só faz sentido porque pode ocorrer de a especificação de um sistema não expressar as necessidades de seus clientes. Por exemplo, essa diferença pode ser causada por um erro na fase de levantamento de requisitos; isto é, os desenvolvedores não entenderam os requisitos do sistema ou o cliente não foi capaz de explicá-los precisamente. A definição de V&V engloba muitas das atividades que são abrangidas pela qualidade de software (SQA).

Conforme SOMMERVILLE (2003), o principal objetivo do processo de V&V é estabelecer confiança de que o software atende ao seu propósito. Assim, o processo de teste de software possui dois objetivos distintos: (i) demonstrar para o desenvolvedor e para o cliente que o software atende aos seus requisitos e (ii) descobrir falhas e defeitos no software onde o comportamento do software está incorreto, indesejável ou não atende a sua especificação. Por exemplo, quando se realiza um teste de um método, para verificar se ele retorna o resultado especificado, estamos realizando uma atividade de verificação. Por outro lado, quando realizamos um teste funcional e de aceitação, ao lado do cliente, isto é, mostrando para ele os resultados e funcionalidades do sistema, estamos realizando uma atividade de validação.

QUESTÃO 2: A complexidade ciclomática é uma métrica de software que fornece uma medida quantitativa da complexidade lógica de um programa, e é medida pela quantidade de caminhos independentes de seus procedimentos ou métodos (PRESSMAN, 2001). Considere o método *Try* fornecido abaixo para identificar os caminhos independentes que determinam sua complexidade. Na resposta desta questão é preciso informar o **valor da complexidade ciclomática** e a **lista dos caminhos independentes** do método. Utilize os números das linhas do código abaixo para identificar a sequência destes caminhos.

```
       public void Try(int[] b){
1        int[] board= new int[3];
2        int attempt = -1;
3        if(b != NULL)
4             board = b;

5        do{
6            do{
7                System.out.print("Line: ");
8                attempt = input.nextInt();

9                if( attempt > 3 ||attempt < 1)
10                    System.out.println("Invalid line. It's 1, 2 or 3");

11           }while( attempt > 3 ||attempt < 1);

12           attempt --;

13           if(board[attempt] != 0)
14               System.out.println("Placed already marked. Try other.");
15     }while(board[attempt] != 0);
     }
```

RESPOSTA: De acordo com PRESSMAN (2001) e SOMMERVILLE (2003), os caminhos de execução independentes de um programa podem ser representados por um grafo de fluxo de controle lógico, onde estruturas de seleção e repetição permitem desvios neste fluxo dando origem a caminhos independentes. Apesar desta representação não ser obrigatória para a resposta desta questão, a Figura 1 ilustra o grafo de fluxo referente ao método acima para melhor descrever a solução.

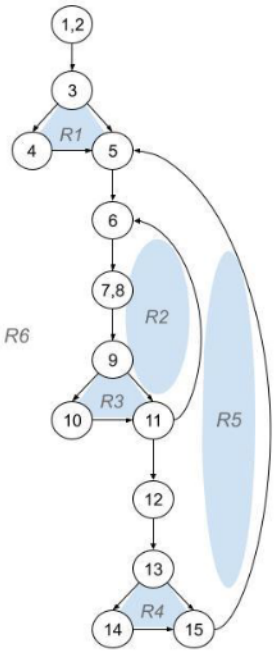


Figura 1 - Grafo de Fluxo

A numeração de linhas de código do método foi utilizada para identificar as instruções no fluxo. Conforme descrito por PRESSMAN (2001), instruções sequenciais e sem estruturas de controle devem ser representadas em um único nó, tal como efetuado para os nós 1,2 e 7,8. As estruturas de seleção (*if*) são representadas pelos nós 3, 9 e 13, enquanto as estruturas de repetição (*do-while*) são representadas pelos fluxos 11-6 e 15-5. De acordo com o mesmo autor, é possível calcular a complexidade ciclomática através de 3 maneiras:

1) O número de regiões do grafo corresponde à complexidade ciclomática. As áreas limitadas por arestas e nós são chamadas de regiões, e estão numeradas na Figura 1. Na contagem de regiões, a área fora do grafo deve ser incluída como uma região. Portanto, neste caso temos 6 regiões no total.
2) A complexidade ciclomática V(G) para um grafo de fluxo G é definida como
   V(G) = E – N + 2
   em que E é o número de arestas do grafo de fluxo e N é o número de nós do grafo de fluxo. Neste caso, E=17 e N=13. Logo, V(G) = 6

3) A complexidade ciclomática V(G) para um grafo de fluxo G também é definida como
V(G) = P + 1
em que P é o número de nós predicados contidos no grafo de fluxo G. Neste caso, temos 3 estruturas de seleção e 2 de repetição com predicados associados, totalizando V(G) = 5 + 1 = 6.

Logo, conclui-se que a complexidade ciclomática desta questão é 6.

Segundo PRESSMAN (2001), um caminho independente é qualquer caminho que introduz pelo menos um novo conjunto de comandos de processamento ou uma nova condição. Quando definido em termos de um grafo de fluxo, um caminho independente deve incluir pelo menos uma aresta que não tenha sido atravessada por outros caminhos já definidos. Logo, a lista de caminhos independentes para esta questão é:

a) 1-2-3-5-6-7-8-9-11-12-13-15
b) 1-2-3-4-5-6-7-8-9-11-12-13-15
c) 1-2-3-4-5-6-7-8-9-10-11-12-13-15
d) 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15
e) 1-2-3-4-5-6-7-8-9-10-11-6-7-8-9-10-11-12-13-14-15
f) 1-2-3-4-5-6-7-8-9-10-11-6-7-8-9-10-11-12-13-14-15-5-6-7-8-9-10-11-6-7-8-9-10-11-12-13-14-15

Observe que cada novo caminho introduz uma nova aresta que se encontra destacada em vermelho.

QUESTÃO 3: A Tabela 1 define um número de tarefas de um projeto de software, suas durações e suas dependências. As dependências especificam que dada tarefa só poderá iniciar quando as suas tarefas dependentes estiverem concluídas. A partir das informações contidas nesta tabela, **desenhe um cronograma** deste projeto, ao final informe qual o **prazo para entrega** do projeto e **quais tarefas estão no caminho crítico** deste projeto. De acordo com PRESSMAN (2001), se as tarefas do caminho crítico sofrerem atraso, o prazo de entrega do projeto é ameaçado. Para desenhar o cronograma utilize um gráfico de *Gantt* ou uma rede de tarefas PERT (*Program Evaluation and Review Technique*). Para o cronograma você poderá utilizar alguma ferramenta para desenho, ou efetuar sua solução em papel. Em ambos os casos um arquivo de imagem em jpg ou png deverá ser gerado para o *upload* na prova. Ao final, **explique o que irá acontecer com os elementos apresentados (caminho crítico e prazo) se a tarefa T4 atrasar 20 dias**, levando o total de 40 dias para ser executada.

Tabela 1

Tarefa	Duração (Dias)	Dependências
T1	10	-
T2	15	T1
T3	10	T1, T2
T4	20	-
T5	10	-
T6	15	T3, T4
T7	20	T3, T4
T8	35	T7
T9	15	T6
T10	5	T5,T9
T11	10	T5,T9

RESPOSTA: De acordo com SOMMERVILLE (2003), o cronograma de um projeto pode ser estabelecido por um gráfico de barras de atividades, também chamado de Gráfico de *Gantt* (PRESSMAN, 2001). Nesta opção estabelece-se um sequenciamento das tarefas baseando-se nas suas respectivas durações e dependências. A Figura 2 apresenta o gráfico de *Gantt* para este projeto. As tarefas em vermelho compõem o caminho crítico deste projeto, visto que suas dependências e durações determinam diretamente o prazo do projeto. Tal prazo é calculado pelo somatório das respectivas durações totalizando 90 dias para este projeto.

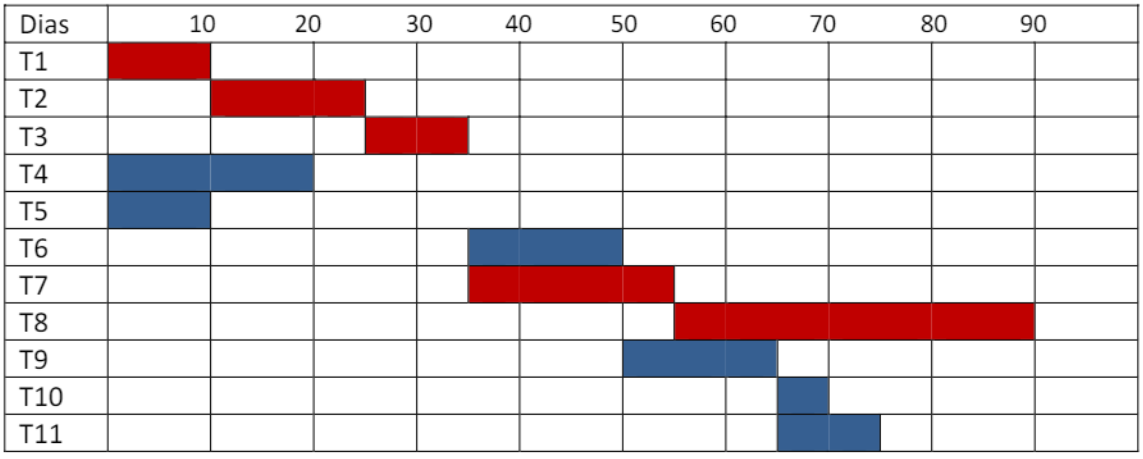


Figura 2 – Gráfico de *Gantt*

A Figura 3 apresenta outra opção de resposta para ilustrar o cronograma, neste caso através de uma rede de tarefas PERT. De acordo com PRESSMAN (2001), uma rede PERT é representada por um grafo em que as

tarefas são unidas de acordo com suas respectivas dependências. O caminho crítico é sinalizado em vermelho. Conforme já descrito anteriormente, o somatório das durações destas tarefas representa o caminho mais longo da rede, que neste caso corresponde a 90 dias.

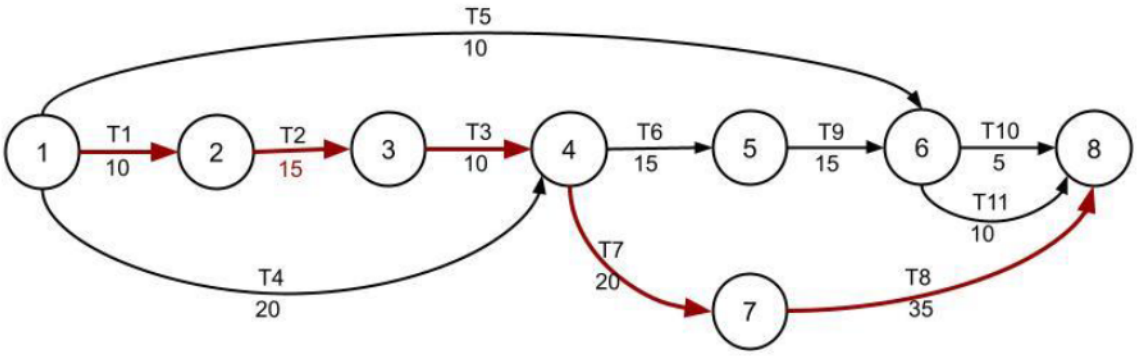


Figura 3 – Rede PERT

Por fim, caso a tarefa T4 seja atrasada em mais 20 dias, totalizando uma duração de 40 dias, o caminho crítico do projeto irá se alterar pois T6 e T7 dependem da conclusão de T3 e T4, e que levarão 35 e 40 dias para serem executadas, respectivamente. Devido ao atraso de T4 sua duração supera a de T3, passando T4 para o caminho crítico. Assim sendo, o caminho crítico ficará T4->T7->T8. O prazo do projeto é também alterado com este atraso, passando para 95 dias (somatório das durações de T4, T7 e T8).

QUESTÃO 4: Cada vez mais se torna difícil ou mesmo impossível prever como um sistema computacional irá evoluir ao longo do tempo. As condições de mercado mudam rapidamente, as necessidades dos usuários se alteram e novas ameaças competitivas emergem sem aviso. Ainda, em muitas situações, não é possível definir completamente requisitos antes que se inicie o projeto de software (PRESSMAN, 2001).

Baseado na afirmação acima, descreva os principais princípios dos métodos ágeis, seus benefícios e dificuldades de utilização. Ainda descreva resumidamente três métodos ágeis para desenvolvimento de software.

RESPOSTA:

Princípios:

Envolvimento do cliente: Os clientes devem estar intimamente envolvidos no processo de desenvolvimento. Seu papel é fornecer e priorizar novos requisitos do sistema e avaliar suas iterações.
Entrega incremental: 0 software é desenvolvido em incrementos com o cliente, especificando os requisitos para serem incluídos em cada um.
Pessoas, não processos: As habilidades da equipe de desenvolvimento devem ser reconhecidas e exploradas. Membros da equipe devem desenvolver suas próprias maneiras de trabalhar, sem processos prescritivos.
Aceitar as mudanças: Deve-se ter em mente que os requisitos do sistema vão mudar. Por isso, projete o sistema de maneira a acomodar essas mudanças.
Manter a simplicidade: Focalize a simplicidade, tanto do software a ser desenvolvido quanto do processo de desenvolvimento. Sempre que possível, trabalhe ativamente para eliminar a complexidade do sistema (SOMMERVILLE, 2003)

Principais benefícios:

Uma das principais características da abordagem ágil é sua habilidade de reduzir os custos da mudança ao longo de todo o processo de software (PRESSMAN, 2001). Os processos de especificação, projeto e implementação são intercalados, não havendo especificação detalhada do sistema e a documentação completa do projeto. O sistema é desenvolvido em uma série de versões, e os usuários finais e stakeholders são envolvidos na especificação e avaliação de cada versão. Os métodos ágeis são métodos de desenvolvimento incrementais em que os incrementos são pequenos e, normalmente, as novas versões do sistema são criadas e disponibilizadas aos clientes a cada duas ou três semanas (SOMMERVILLE, 2003).

Eles são mais adequados ao desenvolvimento de softwares nos quais os requisitos de sistema mudam rapidamente durante o processo de desenvolvimento. Têm como objetivo reduzir a burocracia do processo, evitando qualquer trabalho de valor duvidoso de longo prazo (SOMMERVILLE, 2003). Ainda, um processo ágil deve ser adaptável e incremental, onde a equipe receba feedback do cliente de forma rápida por meio da avaliação de um protótipo operacional. Essa abordagem iterativa capacita o cliente a avaliar o incremento de software regularmente, fornecendo o feedback necessário para a equipe de software e influenciando as adaptações de processo feitas para incluir adequadamente o feedback (PRESSMAN, 2001).

Principais dificuldades:

Difícil de ser utilizado no desenvolvimento de softwares de grande porte e que envolvam sistemas críticos tais como sistemas aeroespaciais e de governo. Também, no desenvolvimento de sistemas que contam com grandes equipes que trabalham de forma distribuída e colaborativa. Nesses casos, onde o software pode

demorar até dez anos desde a especificação inicial até a implantação, seu desenvolvimento envolve um overhead significativo no planejamento, projeto e documentação do sistema. Esse overhead se justifica quando o trabalho de várias equipes de desenvolvimento tem de ser coordenado, quando o sistema é um sistema crítico e quando muitas pessoas diferentes estão envolvidas na manutenção do software durante sua vida. (SOMMERVILLE, 2003)

Exemplos de métodos ágeis para desenvolvimento de software:

Extreme Programming – XP: Emprega uma abordagem orientada a objetos como seu paradigma de desenvolvimento e envolve um conjunto de regras e práticas constantes no contexto de quatro atividades: planejamento, projeto, codificação e testes. O XP define um conjunto de cinco valores que estabelecem as bases para todo trabalho — comunicação, simplicidade, feedback (realimentação ou retorno), coragem e respeito. Cada um desses valores é usado como um direcionador das atividades, ações e tarefas específicas da XP (PRESSMAN, 2001).

Scrum: O desenvolvimento com Scrum incorpora as seguintes atividades estruturais: requisitos, análise, projeto, evolução e entrega. Em cada atividade ocorrem tarefas realizadas dentro de um intervalo de tempo chamado sprint. O trabalho realizado dentro de um sprint é adaptado ao problema em questão e definido, muitas vezes, em tempo real, pela equipe Scrum. O número de sprints necessários para cada atividade varia dependendo do tamanho e da complexidade do produto (PRESSMAN, 2001).

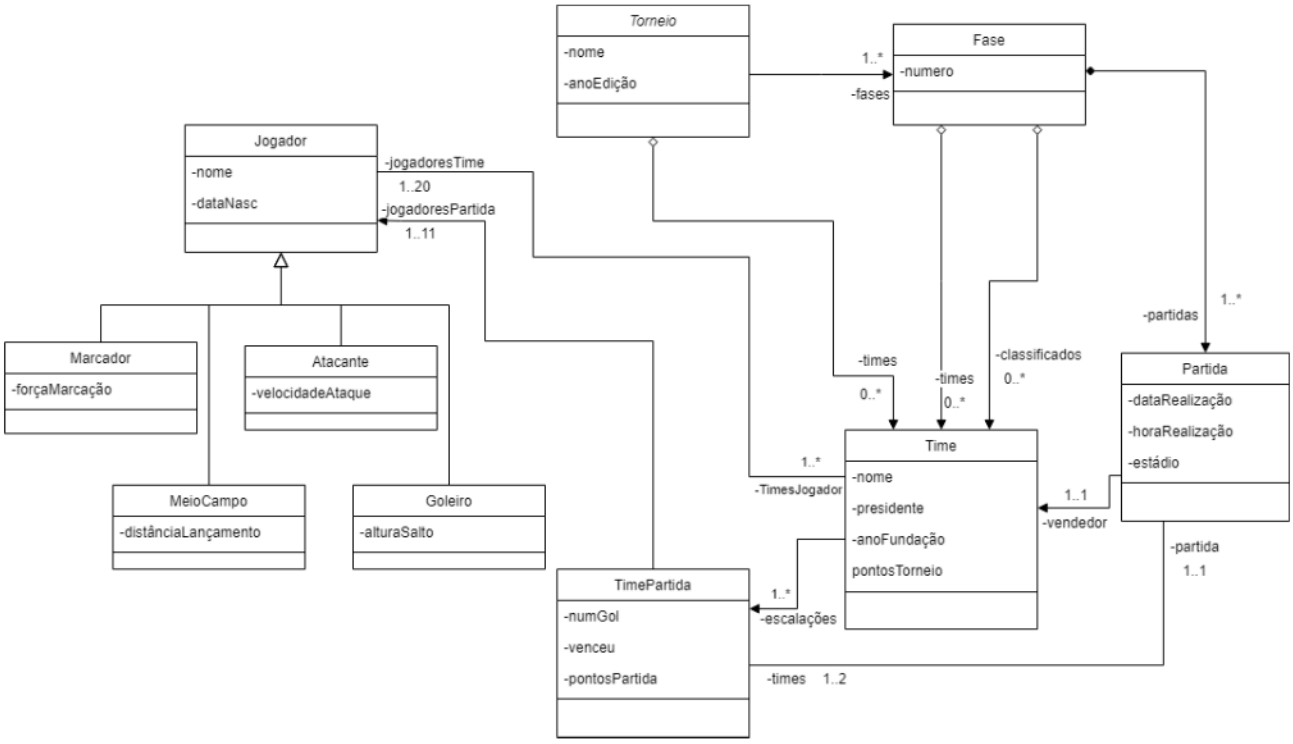
Dynamic Systems Development Method – DSDM: A filosofia DSDM baseia-se em uma versão modificada do princípio de Pareto — 80% de uma aplicação pode ser entregue em 20% do tempo que levaria para entregar a aplicação completa (100%). Ele está estruturado no seguinte ciclo de vida do software: Estudo da viabilidade, Estudo do negócio, Iteração de modelos funcionais, Iteração de projeto e desenvolvimento, e Implementação (PRESSMAN, 2001).

Feature Drive Development – FDD: O FDD adota uma filosofia que enfatiza a colaboração entre pessoas da equipe FDD, gerencia problemas e complexidade de projetos utilizando a decomposição baseada em funcionalidades, seguida pela integração dos incrementos de software, e comunicação de detalhes técnicos usando meios verbais, gráficos e de texto. Ele é composto pelas seguintes etapas: Desenvolvimento do modelo geral, construção de uma lista de funcionalidades, planejamento das funcionalidades, projeto e desenvolvimento das funcionalidades (PRESSMAN, 2001).

Agile Unified Process – AUP: AUP fornece uma camada serial de processo que capacita a equipe a visualizar o fluxo geral do processo de um projeto de software. Entretanto, dentro de cada atividade, a equipe itera ou se repete para alcançar a agilidade e para entregar incrementos de software significativos para os usuários finais tão rapidamente quanto possível. Cada iteração AUP compreende as seguintes atividades: Modelagem, Implementação, Aplicação, Configuração e gerenciamento de projeto, Gerenciamento do ambiente (PRESSMAN, 2001).

QUESTÃO 5: Modele o diagrama de classes para o gerenciamento de um torneio de futebol amador. O torneio é formado por times, sendo que cada time é formado por 20 jogadores, mas apenas uma escalação de 11 jogadores pode entrar em campo. Ainda, cada jogador tem uma posição definida e o time deve ter sempre, 4 marcadores, 4 meio-campos, 2 atacantes e 1 goleiro. Cada tipo de jogador tem algumas características diferentes, como, por exemplo, força de marcação para marcadores, distância de lançamento para meio-campos, velocidade de ataque para atacantes e altura do salto para goleiros. Cada jogador pode jogar em mais de um time e em posições diferentes em cada um deles. O torneio e composto por 8 times que jogam três fases, onde todos se enfrentam e a metade dos times (os melhores) passam para a fase seguinte, sendo que os dois melhores da penúltima fase fazem a final. O time que ganha a partida recebe 3 pontos, o que empata recebe 1 ponto e o que parte recebe 0 pontos.

RESPOSTA:



Referências:

PRESSMAN, R.S. Engenharia de Software. Mc Graw Hill, 5ª ed. 2001.

SOMMERVILLE, I. Engenharia de Software. Addison Wesley, 6ª ed. 2003.

Membros da Banca:

Avaliador 1 – Carla D. M. Berkenbrock	**Avaliador 2 – Éverlin F. C. Marques**
Avaliador 3 – Rebeca S. Freitas	**Presidente da Banca – Fabiano Baldo**