Universidade Federal do Rio Grande do Sul — Instituto de Informática
INF01120 – Técnicas de Construção de Programas — 2019/1
Rafael Baldasso Audibert — 00287695

Lista de Exercícios 2

Modularidade

1. *Considere as técnicas que aprendemos em aulas anteriores, e discuta como elas devem ser empregadas para satisfazer os critérios de modularidade*

   - *Princípios OO:* Conseguimos abusar de técnicas como Classes para aumentar a coesão entre métodos (respeitando o Princípio de Responsabilidade Unica), e Herança e Polimorfismo para tentar evitar acoplamento entre objetos
   - *Tratamento de Exceções e Assertivas:* Tratando os erros dentro da própria classe, estamos evitando o acoplamento entre classes.
   - *Projeto de Software visando Reuso:* Visando o reuso, conseguimos evitar o acoplamento entre as classes, já que teremos um código mais modularizado e separado de acordo com sua funcionalidade.
   - *Convenções e Boas Práticas:* Boas práticas são sempre úteis e melhoraram o desenvolvimento de software, auxiliando também na coesão entre classes.

2. *Uma classe implementa a interface de uma aplicação e também a sua lógica de domínio. Esta classe tem problemas de acoplamento? Tem problemas de coesão? Justifique sua resposta.*
   Possui problemas de coesão já que não existe sentido lógico entre a interface de aplicação e a lógica de domínio. Também não compartilham atributos entre si, já que somente nos métodos com a lógica de domínio os atributos de domínio da classe são usados.

3. *Explique o princípio de single choice. Dê um exemplo de quando não é seguido.*
   O Princípio do Single Choice diz que toda vez que um sistema de software suportar múltiplas alternativas, apenas um módulo do sistema deve conhecer essas alternativas, assim conseguimos abstrair as necessidades do código, fazendo com que caso adicionemos mais alguma alternativa, por exemplo, possamos alterar isso em somente um lugar. Para utilizarmos desse princípio, técnicas de orientação a objetos como Herança e Polimorfismo auxiliam bastante.
   Um exemplo de quando isso não é seguido é quando, por exemplo, temos mais do que uma interface para o mesmo sistema (para diferentes usuários, por exemplo). Se armazenarmos todas as opções disponíveis em todas as UI's, precisaremos iterar sobre todas elas para adicionarmos uma nova função. Se todas funções forem armazenadas organizadamente em somente um módulo, a manutenção e consequentemente adição de novas funções às interfaces se torna imensamente mais fácil.

4. *Relacione a regra de modularidade "Interfaces Pequenas" com o "Princípio da Responsabilidade Única".*
   A regra de Interfaces pequenas serve para que possamos ter um baixo acoplamento, fazendo com que os canais de comunicação entre diferentes classes seja o mínimo possível. O SRP busca fazer com que cada classe tenha um único propósito, evitando o acoplamento, e concordando com a regra de Interfaces Pequenas, já que com uma única responsabilidade, a quantidade de dados passado entre classes é pequena.

5. *Considere o seguinte código, e responda qual o nível de coesao entre os métodos da classe Utils?*

```
class Utils {
    private Utils() { }
    public static String dateToString() { ...}
    public static boolean validateNumber(...) { ...}
    public static int calculateStringDistance(...) { ...}
    public static Connection connectDatabase(...) { ...}
}
```

   Nível Lógico, já que as funções não fazem sentido umas com as outras, nem compartilham das mesmas estruturas

6. *Considere o seguinte código e responda qual o nível de acoplamento entre o método fazAlgo e o método sort?*

```
class Cliente {
    public String fazAlgo() {
        helper.sort(list, true);
    }
}

class Helper {
    public void sort(List list, boolean bubbleSort) {
        if (bubbleSort)
            // bubble sort
        else
            // quicksort
    }
}
```

   Nível de controle, já que passamos uma flag para podermos escolher qual tipo de sort será executado.

SQA

1. *Qual a diferença entre validação e verificação?*
   Verificação é o processo de se avaliar um software a cada fase para determinar se o produto dessa fase satisfaz ao que foi requerido no início da fase, respondendo a pergunta ”Estamos desenvolvendo certo o produto?”.
   Validação é o processo de se avaliar um software, durante ou após o desenvolvimento, para determinar se o produto satisfaz aos requisitos, respondendo a pergunta ”Estamos desenvolvendo o produto certo?”

2. *O que significa fazer uma verificação estática?*
   Significa fazermos uma verificação do software sem executarmos o produto, visando determinar propriedades do produto válidas para qualquer execução do produto final. Para isso utilizamos inspeções do programa, verificação baseada em matemática ou modelos, analisadores estáticos de programas ou utilizando desenvolvimento de software ”Clean Room”.

Teste de Software

1. *O que caracteriza um bom caso de teste?* Aquele que tem alta probabilidade de revelar erro ainda não descoberto, podendo revelar especificações incompletas ou ambíguas.

2. *Técnicas de teste (caixa branca e caixa preta) devem ser utilizadas em níveis de teste (unidade, integração, sistema, aceitação) específicos? Explique sua resposta.*

Teste Funcional

1. *Dado um cenário de busca de CEP, cuja entrada sejam 5 dígitos de CEP (padrão americano) e como saída temos uma lista de cidades. Quais são os valores representativos (ou classes de valores) para teste?*
   tamanho_entrada = 5

2. *Dado o cenário abaixo defina as classes de equivalência e os casos de teste:*

   - *Procedimento Valida_Nova_Senha que recebe como entrada uma senha e valida-a conforme as seguintes regras*
     - *Uma senha deve ter de 6 e 10 caracteres*
     - *O primeiro caractere deve ser alfabético, numérico ou um caractere de "?"*
     - *os outros caracteres podem ser quaisquer, desde que não sejam caracteres de controle*
     - *a senha não pode existir em um dicionário*

   Classes de equivalência:

   - tamanho_entrada $\in$ [6, 10]
   - primeiro_caractere $\in$ [a-zA-Z0-9\?]
   - outros_caracteres $\notin$ [caracteres_de_controle]
   - senha $\notin$ lista_palavras_pre_definidas

   Casos de teste:

   - Palavra que é aceita (está dentro de todos os valores aceitáveis) - Deve ser ACEITA
   - Palavra com tamanho menor do que 6 - Deve ser RECUESADA
   - Palavra com tamanho igual a 6 - Deve ser ACEITA
   - Palavra com tamanho igual a 10 - Deve ser ACEITA
   - Palavra com tamanho maior do que 10 - Deve ser RECUSADA
   - Palavra com primeiro caractere alfabético - Deve ser ACEITA
   - Palavra com primeiro caractere numérico - Deve ser ACEITA
   - Palavra com primeiro caractere "?" - Deve ser ACEITA
   - Palavra com primeiro caractere especial - Deve ser RECUSADA
   - Palavra sem caracteres de controle - Deve ser ACEITA
   - Palavra com caracteres de controle - Deve ser RECUSADA

- Palavra que não esteja no dicionário - Deve ser ACEITA
- Palavra que esteja no dicionário - Deve ser RECUSADA
- Palavra que esteja no dicionário com uma capitalização diferente - Deve ser RECUSADA
- Palavra igual a NULL - Deve ser RECUSADA, sem ter erros

Métricas de Software

1. *Você desenvolveu um sistema e coletou métricas de software dele. Como resultado, você obteve a tabela abaixo. Aponte uma classe que potencialmente deveria ser refatorada com base nestes resultados. Explique sua resposta.*

Classe	CBO	NA	NO	DIT	LOC	WMC
A	21	14	17	4	184	17
B	27	7	15	3	261	21
C	47	13	28	2	543	41
D	14	8	13	1	205	24

CBO (*Coupling between Object Classes*): acoplamento entre classes de objetos
NA (*Number of Attributes*): número de atributos
NO (*Number of Operations*): número de operações
DIT (*Depth of Inheritance Tree*): profundidade da árvore de herança
LOC (*Lines of Code*): número de linhas de código
WMC (*Weighted Methods per Class*): número de métodos da classe ponderados pela complexidade de cada método

Classe C. Provavelmente deveria ser separada em mais do que uma classe pelos seguintes motivos:

- CBO elevado (em relação aos outros) implica que a classe está se comunicando com muitas outras classes, o que faz com que o acoplamento seja muito alto.
- NO elevado (em relação aos outros) implica que a classe possuí muitos métodos, o que pode ocasionar na perca da coesão.
- LOC está elevado (em relação aos outros) mesmo se levarmos em consideração que ele possui um NO maior, portanto seus métodos estão compridos demais implicando na falta do ”Princípio de Responsabilidade única”.
- WMC elevado (em relação aos outros) completa mais ainda a afirmação anterior de que os métodos provavelmente estão fazendo coisas demais

Refatoração

1. *Considere as três versões apresentadas uma classe Group que tem um método chamado getUsers(). Ordene as três versões em relação a sua qualidade (de melhor para pior).*

   - Versão 1

```
public class Group…
   public List getUsers() {
      List users = new ArrayList();

      if (!userDirectoryExists())
         return users;

      // Add found users to users collection
      File[] files = new File(persistencePath()).listFiles();
      for (File file : files)
         if (file.isDirectory())
            users.add(new User(file.getName(), this));

      // sort by most recently registered users
      Collections.sort(users, new User.UserComparatorByDescendingRegistration());

      return users;
   }
```

   - Versão 2

```
public class Group…
   // Gets users sorted by the most recently registered user
   public List getUsers() {
      List users = new ArrayList();

      if (!new File(persistencePath()).exists())
         return users;

      File[] files = new File(persistencePath()).listFiles();
      for (File file : files)
         if (file.isDirectory())
            users.add(new User(file.getName(), this));

      Collections.sort(users, new User.UserComparatorByDescendingRegistration());

      return users;
   }
```

- Versão 3

```
public class Group…
   public List getUsersSortedByMostRecentlyRegistered() {
      List users = new ArrayList();

      if (!userDirectoryExists())
         return users;

      addFoundUsersTo(users);

      sortByMostRecentlyRegistered(users);

      return users;
   }
```

De melhor para pior: 3, 1, 2.

2. *Discuta que refatoração deve ser feita para levar a pior versão para a melhor.* Empregar o uso do ”Princípio de Reponsabilidade única” fazendo com que tarefas específicas sejam executadas cada uma em seu código, como a verificação de existência de arquivo, a criação da lista de arquivos e a ordenação dos mesmos.

3. *Sugira uma refatoração do seguinte método, para que ele tenha uma única responsabilidade. Mostre como seria a nova versão de código.*

```
1   public bool Login(string username, string password)
2   {
3       var user = userRepo.GetUserByUsername(username);
4
5       if(user == null)
6           return false;
7
8       if (loginValidator.IsValid(user, password))
9           return true;
10
11      user.FailedLoginAttempts++;
12
13      if (user.FailedLoginAttempts >= 3)
14          user.LockedOut = true;
15
16      return false;
17  }
```

Abstrairia-se da 7ª linha em diante em um método que deveria pertencer ao próprio user, como representado a seguir:

```
1   public bool Login(string username, string password)
2   {
3       var user = userRepo.GetUserByUsername(username);
```

```
4
5           if(user == null)
6               return false;
7
8           return user.isValid(password);
9       }
```