Código Limpo Intermediário

Acoplamento

- Um dos objetivos de planejar o *software* é garantir o menor acoplamento possível entre seus componentes.
- O acoplamento é o grau de interdependência entre os componentes de um *software*.
- Um alto grau de acoplamento significa que os objetos não estão se comunicando a nível de *interface*, mas acessando diretamente a implementação de outros objetos e módulos.
- Cada mudança em um componente pode levar a mudança em outros componentes.

Lei de Deméter

- Em 1987, Ian Holland propôs a Lei de Deméter ou Princípio do Menor Conhecimento.
- Este princípio diz, de maneira geral, que um objeto a não deve acessar um objeto c através de um objeto b.
- Abaixo, segue um exemplo de violação deste princípio:

```
public void accessC() {
  String value = this.b.doSomething().getValue();
}
```

- No exemplo acima, o objeto dono método accessC, está chamando um método de um objeto b (`doSomething`) e, a partir deste retorno, está acessando um método de um objeto c que é retornado por `doSomething`.

Lei de Deméter

A Lei de Deméter diz que um método `m` de um objeto `a` só pode invocar métodos dos seguintes objetos:

- do próprio `a`;
- dos parâmetros de `m`;
- dos objetos instanciados em `m`;
- dos atributos de `a`;
- de variáveis globais acessíveis por `a` e no escopo de `m`.

Lei de Deméter

- A Lei de Deméter também pode ser renomeada para "Lei do Um Ponto".
- Isto significa que não deve haver dois ou mais pontos em um termo (exclui-se da conta o `this.`).
- No exemplo anterior, o objeto `b` deveria ter um método que encapsule a chamada a `getValue` do objeto retornado em `doSomething`.
- A diminuição do acoplamento, neste caso, se dá pelo fato de que `a` não precisará saber os detalhes de implementação de `b`.

Lei de Deméter

- Entretanto, deve-se tomar cuidado com a aplicação da Lei de Deméter.
- Tenha-se o exemplo abaixo[1]:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

E sua refatoração:

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

- Isso pode levar a uma explosão na quantidade de métodos dentro de `ctxt`[2].

[1] Retirado de Martin, 2009.

[2] Classes muito grandes não são bom sinal.

Lei de Deméter

- A refatoração anterior esconde falsamente a implementação de `ctxt`.
- O grande problema desta refatoração é que `getAbsolutePathOfScratchDirectoryOption` é uma consulta por um caminho para um diretório e não a ação desejada.
- O que se deseja é criar um arquivo no diretório.
- Seria mais interessante que `ctxt` tivesse um método `createScratchFileStream`.

Tratamento de exceções

- Uma das vantagens da utilização de linguagens orientadas a objetos é a capacidade de levantar e capturar exceções.
- As exceções substituem os antigos e nem tão elegantes códigos de erro.
- Entretanto, apenas usar exceções por usar não faz sentido.
- Elas devem ser escritas dentro de um contexto claro e que tornem o código ainda mais limpo.

Tratamento de exceções

- Uma técnica bastante útil para limpar o código é encapsular em uma função o tratamento de uma dada exceção.
- Abaixo, tem-se um exemplo[3]:

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}

private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
}
```

[3] Retirado de Martin, 2009.

Tratamento de exceções

- Um tipo especial de código de erro é retornar `null` quando não uma consulta retorna vazio.
- Neste caso, a função que chama a consulta deve testar se o retorno é diferente de `null`.
- Em métodos onde há apenas uma consulta, o código pode não ficar tão longo, mas imaginemos que são feitas 4 consultas diferentes dentro de um método. Isto resultará em 4 `ifs` aninhados.
- Neste caso, é interessante que os métodos de consulta retornem exceções.
- Caso queiramos uma lista ou outra estrutura de dados similar, linguagens como Java permitem criar listas vazias.

Tratamento de exceções

Tenha-se o exemplo abaixo[4]:

```
List<Employee> employees = getEmployees();
  if (employees != null) {
    for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

Se `getEmployees` retornar uma lista nula ao invés de `null`, o código poderá ser escrito assim:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
  totalPay += e.getPay();
}
```

[4] Retirado de Martin, 2009.

Encapsular APIs de terceiros

- Qualquer sistema grande, em alguma parte, utilizará alguma biblioteca ou serviço de terceiros.
- Em teoria, isto torna o desenvolvimento mais rápido e seguro[5].
- Por outro lado, cria uma dependência do projeto em relação a essa API.
- Para mitigar este problema, pode-se encapsular essa API.

[5] Há controvérsias. . . , mas não está no nosso escopo discutí-las.

Encapsular APIs de terceiros

Tem-se o exemplo abaixo:

```
ACMEPort port = new ACMEPort(12);
try {
  port.open();
} catch (DeviceResponseException e) {
  reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
  reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
  reportPortError(e);
  logger.log("Device response exception");
} finally {
  ...
}
```

Encapsular APIs de terceiros

Pode-se encapsular `ACMEPort` em uma classe chamada LocalPort que encapsulará `ACMEPort`, tratando as exceções e dando uma *interface* mais de alto nível[6].

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {
  ...
}
```

- Este código é muito mais limpo que o anterior.
- Além disso, ele trás a vantagem de que poderemos trocar ACMEPort por outra API sem necessitar modificar a aplicação toda, apenas a classe `LocalPort`.

[6] Exemplo retirado de Martin, 2009.

Encapsular APIs de terceiros

Segue a implementação de `LocalPort`[7]:

```
public class LocalPort {
  private ACMEPort innerPort;

  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }

  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
      throw new PortDeviceFailure(e);
    }
  }
}
```

Observe também que LocalPort lança uma exceção muito mais "amigável".

[7] Adaptado de Martin, 2009.

Testes automatizados

- Apesar de ser uma prática já antiga, parte dos desenvolvedores, especialmente os iniciantes, não escreve testes automatizados para seu código.
- Um teste automatizado é um programa que testa o programa.

Testes unitários

- Dentre os diversos tipos de testes possíveis, o mais simples (porém não menos importante), é o teste unitário (ou teste de unidade).
- Um teste unitário geralmente é composto por uma classe, onde cada método é um teste.
- Cada teste deve testar um método de uma classe de produção.
- Testar significa passar um conjunto de dados de entrada e verificar se a saída (já conhecida) é correta.

Testes unitários

Abaixo, um exemplo de teste unitário para os métodos de uma classe `Rectangle` em Java:

```
public class RectangleTests {
  private final Rectangle rect = new Rectangle(10, 10);

  @Test
  public void testCalculateArea() {
      assertEquals(rect.calculateArea(), 100);
  }

  @Test
  public void testCalculatePerimeter() {
      assertEquals(rect.calculatePerimeter(), 40);
  }
}
```

Testes automatizados

- Os testes, porém não servem apenas para garantir que o código funciona.
- Eles servem para dar segurança ao programador.
- Não é incomum times de desenvolvimento evitarem mexer em código de produção por medo de "quebrá-lo".
- Porém, se o código é coberto por testes, basta realizar uma mudança e executar os testes. Se algum erro for detectado, pode-se corrigí-lo ou simplesmente realizar o *rollback* do código a um ponto estável.

Testes limpos

- Dentro do movimento ágil, os testes automatizados se tornaram cidadãos de primeira classe.
- Isto significa o código do testes deve (ou deveria) estar de acordo com os mais altos padrões de qualidade.
- Os testes, assim como código de produção, também mudam com o tempo.
- Eles devem refletir também as necessidades de negócio.
- Como não há testes dos testes, tem-se uma situação bastante sensível porque pode haver alguma insegurança em mexer nos testes.
- Por isso, é necessário que o código dos testes seja extremamente limpo.

Testes limpos

- Testes difíceis de ler podem ser um grande problema.
- Tenha-se o exemplo abaixo[8]:

```
public void testGetPageHieratchyAsXml() throws Exception
{
  crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response = (SimpleResponse)
    responder.makeResponse(new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
}
```

Note como esse teste é um tanto complicado de ler.

[8] Adaptado de Martin, 2009.

Testes limpos

- Após a refatoração abaixo, ele torna-se muito mais fácil de ser lido[9]:

```
public void testGetPageHierarchyAsXml() throws Exception {
  makePages("PageOne", "PageOne.ChildOne");

  submitRequest("root", "type:pages");

  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>", "<name>ChildOne</name>"
  );
}
```

- Agora é muito mais simples entender o que se passa no teste.
- O padrão adotado por esse teste é conhecido como BUILD-OPERATE-CHECK.

[9] Adaptado de Martin, 2009.

Testes limpos

- Outro ponto interessante sobre testes é que eles permitem ao leitor conhecer o código de produção sem lê-lo.
- Tenha-se um dos testes da classe `Rectangle`:

```
private final Rectangle rect = new Rectangle(10, 10);

@Test
public void testCalculateArea() {
    assertEquals(rect.calculateArea(), 100);
}
```

- Mesmo sem conhecer o código de `Rectangle`, sabe-se como `calculateArea` deve comportar-se.

F.I.R.S.T

Martin sugere que os testes sigam regras de acordo com o acrônimo F.I.R.S.T:

- **F**ast - os códigos não devem rodar rapidamente. Se demorarem muito, serão rodados menos frequentemente.
- **I**ndependent - um teste não deve depender de outro.
- **R**epeatable - os testes devem ser repetíveis em qualquer ambiente (produção, QA, desenvolvimento).
- **S**elf-Validating - os testes devem ter uma saída lógica (verdadeiro ou falso).
- **T**imely - os testes devem ser escritos antes do código em produção.

Referências

- Martin, Robert Cecil. Clean Code: A Handbook of Agile Software Craftsmanship. 2009. Prentice Hall.
- Wikipedia. Demeter Law. Acessado em: 23 outubro de 2023.