Clean Code:

A Handbook of Agile Software Craftsmanship

Chapter 07: Error Handling

Error Handling

A key aspect of maintaining clean code is how we handle errors and exceptions. Occasionally, the input might be invalid, resulting in an exception being thrown. As programmers, it's our responsibility to ensure that even when faced with unexpected input, our software behaves correctly. However, handling exceptions should not disrupt the program's logic. Effective code is code that handles errors beautifully. To achieve this, there are a few important points to consider.

Point 1

- Prefer exceptions over error codes.

  In older programming languages, exceptions weren't available. Instead, programmers often used boolean variables set to true or returned enum values as error codes within methods. Error handling was then checked during method calls. A major issue here was that programmers could forget to handle errors since the compiler didn't enforce error handling. Nowadays, we use exceptions, which ensure that errors are dealt with, as the compiler generates error messages if we neglect them.

Point 1

```
public void method(){
   doSomething();
   if(anythingWentWrong())
      error = ErrorCode.SomethingWentWrong;
   else if(runningFailed())
      error = ErrorCode.RunningFailed;
   else
      error = ErrorCode.EverythingWasOK;
}
```

```
public void method() {
   doSomething();
   if (anythingWentWrong() || runningFailed())
      errorOccured = true;
}
```

```
public void method() throws SomeException {
   doSomething();
   if (somethingWentWrong() || runningFailed())
      throw new SomeException();
}
```

Point 2

- Commence methods with try-catch-finally blocks.

  Using **try-catch-finally** divides our code into three blocks. The code within the **try** block might encounter an error. If an error occurs, the compiler stops executing subsequent lines and enters the **catch** block to manage the exception. After the **try-catch** block executes, the **finally** block runs and performs necessary actions. This structure enhances code readability and comprehensibility. Note that this structure should be placed within a method, where it handles the exception. Consequently, we can call this method whenever required. Attempt to write a test that triggers the exception to ensure proper handling.

Point 2

```
public static FXMLFile openPage(String pageName) throws FileLoadException {
   fxmlFile = loadSomeFXMLFile ();
   if (fxmlFile == null)
      throw new FileLoadException();
}
```

```
private void openSomePage() {
   try {
      fxmlFile = openPage(”pageName”);
   } catch (FileLoadException ex) {
      showMessage("The page could not be opened");
   }
}
```

Point 3

- Favor unchecked exceptions.

  Checked exceptions are advantageous for error handling, but they come with a drawback. If a method declares an exception in its signature and it's utilized in higher-level methods, the exception must be included in the signatures of all methods that use it, unless placed within a **try-catch** block. This violates the **Open-Closed Principle (OCP)** of **SOLID** principles. Altering a method in a lower software layer necessitates modifying the signatures of all corresponding methods in higher layers. This approach also conflicts with encapsulation, as higher-level methods gain insight into the exceptions of lower-level methods.

Point 4

- Enhance exceptions with context.

  Pass sufficient information about the exception to enable appropriate reactions when encountering an error. Crucial details to provide include the cause and type of the exception.

Point 4

```
public static void openPage(String pageName) throws FileLoadException {
   fxmlFile = openSomeFXMLFileAndLoadAPage();
   if(fxmlFile == null)
      throw new FileLoadException("Loading the FXML file caused this IOException.");
}
```

Point 5

- Call exception packages based on caller's necessities.

  In some cases, multiple exceptions can arise within a single method, and they all should be appropriately handled.

Point 5

```
public static void openPage(String pageFile) throws FileLoadException, FileAccessDeniedException, NoSuchFXMLFileException {
   if(! pageFile.exits())
      throw new NoSuchFXMLFileException("FXML file did not exist and opening it caused NullPointerException");

   if(! pageFile.isAccessable())
      throw new FileAccessDeniedException("FXML file was not in an accessible directory and opening it caused IOException.");

   fxmlFile = openSomeFXMLFileAndLoadAPage(pageFile);
   if(fxmlFile == null)
      throw new FileLoadException("Loading the FXML file caused IOException.");
}
```

Point 6

- Avoid Returning Null.

Using null as a return value can lead to an increased risk of a **NullPointerException**, which is a complex exception to handle. A better alternative is to return an empty or default object. The optimal approach is to define custom exceptions and throw them instead of returning **null**.

Point 6

```
public Password getPasswordFromUser() {
   Password result;

   if (isPasswordConfirmed())
      result = password;
   else
      result = null;

   return result;
}
```

```
public Password getPasswordFromUser() throws PasswordNotConfirmedException{
   Password result;

   if (isPasswordConfirmed())
      result = password;
   else
      throw new PasswordNotConfirmedException();

   return result;
}
```

Point 7

- Avoid Passing Null.

While returning **null** in methods is problematic, passing **null** to methods is even worse. Although we could check for null values within the method, there's a high chance of overlooking some cases, resulting in potential **NullPointerExceptions** – which are challenging to deal with. The most effective method is to verify the value before calling the method, thus preventing the passing of **null**.

Summary

Clean code is not only readable but also robustness. By adhering to the points we've discussed, we can craft clean, powerful code. This approach allows us to separate business logic from error handling, ultimately enhancing the software's maintainability.