Clean Code:

A Handbook of Agile Software Craftsmanship

Chapter 07: Michael Feather’s Error Handling

Error Handling

Part of having a clean code depends on how we handle the errors and exceptions. Since the input is sometimes an invalid value that leads to an exception thrown. As a programmer we should make sure that even the input is something unexpected, our software does everything fine. But exception handling should not interrupt the program's logics.

A powerful code is a code that handles the errors beautifully. To have such code we should pay attention to a few points.

Point 1

- Use exceptions instead of returning error codes.

  In the old programming languages there were no exceptions; so the programmers used to set a boolean variable **true** or return an **enum** value as error code inside the method. Then they check for errors on method calling. The biggest problem here is that the programmer would have forgotten to handle the errors, since the complier did not check the error handling. Now we use exceptions which makes us sure that the errors are handled. Because the complier shows us error messages if we don't.

Point 1

❌

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

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

✓

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

Point 2

- Start the method with try-catch-finally.

  Using **try-catch-finally** creates three blocks in our code. Codes in **try** block could encounter an error. If so, the compiler stops running next lines of code and enters the **catch** block to handle the exception. After the execution of **try-catch** block the **finally** block runs and does what should be done. This structure makes the code look clean and recognized. Note that this structure should be placed inside a method and it handles the exception. Then we can call this method where ever we want.

  Try to write a test that throws the exception so you could be sure that you handled it properly.

Point 2

```
public static FXMLFile openPage(String pageName) throws FileLoadException {
   fxmlFile = openSomeFXMLFile ();
   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

- Use unchecked exceptions.

  Checked exceptions are a great advantage in handling the errors. But they come with a cost. The cost is that if a method throws an exception in its signature and it is used in upper levels, the exception has to be added to the signature of all methods that use it, until we put it inside a **try-catch** block. This violates Open-Closed-Principle(OCP) from S.O.L.I.D principles. Because changing one of the methods in one of lower layers of the software forces us to change the signature of all those methods in the upper levels. It also has a conflict with encapsulation. Because the methods of upper levels have information about the exceptions of methods in lower levels.

Point 4

- Provide context with exceptions.

  Pass enough information about the exception to be able to react properly in case of encountering the error. The most important information to be provided are the cause and the type of the exception

Point 4

```java
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 callers necessities.

  Multiple exceptions may be generated inside one method and they all should be 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

- Don’t return null.

Sometimes we return null when the code encounters an error. It increases the risk of NullPointerException which is a difficult exception to handle. The first alternative way is to return an empty object with default values. The best alternative way is to define customized 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() {
   Password result;
   if (isPasswordConfirmed())
      result = password;
   else
      throw new PasswordNotConfirmedException();

   return result;
}
```

✓

Point 7

- Don't pass null.

Returning **null** in methods is bad; passing **null** to methods is worse. We could check the arguments inside the method for **null** value. But it is very likely that we forget some cases and face **NullPointerException** which is a little tricky to handle. The best way is to check the value before the method call and avoid passing **null** to the method.

Summary

Clean code is not only readable, but also powerful. We can write powerful clean code if we observe these points that we argued about. It provides the terms to isolate business logics from error handling which increases the maintainability of the software.