Review code refactoring scenarios and GitHub Copilot best practices

6 minutes

Refactoring code is the process of restructuring existing code without changing its behavior. The benefits of refactoring include improving code readability, reducing complexity, making the code easier to maintain, and allowing new features to be added more easily. Implementing GitHub Copilot best practices can help you work more efficiently and get better results.

GitHub Copilot best practices

GitHub Copilot is a powerful tool that can help you to refactor and improve your code. However, to get the best results, you should follow some best practices when using Copilot.

Improve responses with better prompts

You can enhance the quality of Copilot's responses by writing more effective prompts. A well-crafted prompt can help Copilot understand your requirements better and generate more relevant code suggestions.

The following guidelines can help you to write better prompts:

- Start general, then get specific.

```plaintext
Generate a Calculator class. Add methods for addition, subtraction, multiplication, division, and factorial. Don't use any external libraries and don't use recursion.
```

- Give examples of what you want.

```plaintext
Generate a function that takes a string and returns the number of vowels in it. For example: findVowels("hello") returns 2, findVowels("sky") returns 0.
```

- Break down complex tasks into simpler tasks.

  Instead of asking Copilot to "Generate a meal planner app", break your prompts down into smaller tasks.

  For example:

  ```plaintext
  Generate a function that takes a list of ingredients and returns a list of recipes.
  ```

  ```plaintext
  Generate a function that takes a list of recipes and returns a shopping list.
  ```

  ```plaintext
  Generate a function that takes a list of recipes and returns a meal plan for the week.
  ```

- Provide the right context, such as code selections, files, terminal output, and more.

  Example, use the `#codebase` variable to refer to the entire codebase: "Where is the database connection string used in #codebase?"

- Iterate on your prompts.

  Provide follow-up prompts to refine or modify the response.

  For example, if you start with the following prompt:

  "Write a function to calculate the factorial of a number."

  You can follow up with:

  "Don't use recursion and optimize by using caching."

  And then:

  "Use meaningful variable names."

- Keep chat history relevant.

Copilot uses history of the conversation to provide context. Remove past questions and responses from the history if they're not relevant. Or, start a new session if you want to change the context.

Provide the right context and tools

Enrich your prompts with relevant context to get more accurate and relevant responses in chat. With the right tools, you can boost your developer productivity.

- In agent mode, select the tools button to configure the tools you want to use or explicitly add then to your prompt.
- Use `#codebase` to let Copilot find the right files automatically by performing a code search.
- Use the `#fetch` tool to fetch content from a web page or use `#githubRepo` to perform a code search on a GitHub repository.
- Reference files, folders, or symbols in your prompt by using `#<file name>`, `#<folder name>`, or `#<symbol>`.
- Drag and drop files, folders, or editor tabs onto the chat prompt.
- Add problems, test failures, or terminal output to your chat prompt for scenario-specific context.

> **Note**
>
> When you use agent mode, Copilot autonomously finds the relevant files and context for you.

Common code refactoring scenarios

GitHub Copilot can help you to quickly refactor code in many ways. The following list includes some common scenarios for refactoring code:

- Optimizing inefficient code
- Cleaning up repeated code
- Making code more concise
- Splitting up complex units of code
- Rewriting conditional code for better readability
- Reformatting code to use a different structure

Use GitHub Copilot to optimize inefficient code

Copilot can help you to optimize code. For example, you can optimize your code to make the code run more quickly.

Consider the following bash script:

Bash

```
#!/bin/bash
# Find all .txt files and count the number of lines in each file
for file in $(find . -type f -name "*.txt"); do
  wc -l "$file"
done
```

You can use chat features to evaluate options for optimizing different aspects the script, such as performance.

Bash

```
#!/bin/bash
# Find all .txt files and count the number of lines in each file (improved performance)
find . -type f -name "*.txt" -print0 | xargs -0 wc -l
```

Use GitHub Copilot to clean up repeated code

Avoiding repetition will make your code easier to revise and debug. For example, if the same calculation is performed more than once at different places in a file, you could move the calculation to a function.

In the following very simple JavaScript example, the same calculation (item price multiplied by number of items sold) is performed in two places.

JavaScript

```
let totalSales = 0;
let applePrice = 3;
let appleSold = 100;
totalSales += applePrice * appleSold;

let orangePrice = 2;
let orangeSold = 50;
totalSales += orangePrice * orangeSold;

console.log(`Total: ${totalSales}`);
```

You can ask chat features to move the repeated calculation into a function.

JavaScript

```
function calculateTotal(price, quantity) {
    return price * quantity;
}
let totalSales = 0;
let applePrice = 3;
let appleSold = 100;
totalSales += calculateTotal(applePrice, appleSold);

let orangePrice = 2;
let orangeSold = 50;
totalSales += calculateTotal(orangePrice, orangeSold);
console.log(`Total: ${totalSales}`);
```

Use GitHub Copilot to make code more concise

If code is unnecessarily verbose it can be difficult to read and maintain. Copilot can suggest a more concise version of selected code.

The following Python code outputs the area of a rectangle and a circle, but the code could be written more concisely:

Python

```
def calculate_area_of_rectangle(length, width):
    area = length * width
    return area

def calculate_area_of_circle(radius):
    area = 3.14 * radius * radius
    return area

rectangle_length = 5
rectangle_width = 10
rectangle_area = calculate_area_of_rectangle(rectangle_length, rectan-
gle_width)
print(f"Area of rectangle: {rectangle_area}")

circle_radius = 7
circle_area = calculate_area_of_circle(circle_radius)
print(f"Area of circle: {circle_area}")
```

GitHub Copilot can help you to refactor the code and make it more concise.

Python

```
def calculate_area_of_rectangle(length, width):
    return length * width

def calculate_area_of_circle(radius):
    return 3.14 * radius * radius

rectangle_length = 5
rectangle_width = 10
print(f"Area of rectangle: {calculate_area_of_rectangle(rectangle_length,
rectangle_width)}")

circle_radius = 7
print(f"Area of circle: {calculate_area_of_circle(circle_radius)}")
```

Use GitHub Copilot to split up complex units of code

Large methods or functions that perform multiple operations are likely to offer fewer opportunities for reuse than smaller, simpler functions that are focused on performing a particular operation. They may also be more difficult to understand and debug.

Copilot can help you to split up complex blocks of code into smaller units that are more suitable for reuse.

The following Python code is a very simple example, but it shows the principle of splitting up a single function into two functions that perform particular operations.

Python

```
import pandas as pd
from pandas.io.formats.styler import Styler

def process_data(item, price):
    # Cleanse the data
    item = item.strip()    # Remove leading and trailing whitespace
    price = price.strip()  # Remove leading and trailing whitespace
    price = float(price)   # Convert price to a float
    # More cleansing operations can be added here

    # Create and print a DataFrame
    data = {'Item': [item], 'Price': [price]}
    df = pd.DataFrame(data)
    print(df.to_string(index=False))

# Example usage
item = "  Apple  "
price = "  1.50  "
process_data(item, price)
```

You can use Copilot to refactor the code to create functions for cleansing data, printing data, and processing data.

Python

```python
import pandas as pd
from pandas.io.formats.styler import Styler

def cleanse_data(item, price):
    # Cleanse the data
    item = item.strip()    # Remove leading and trailing whitespace
    price = price.strip()  # Remove leading and trailing whitespace
    price = float(price)   # Convert price to a float
    return item, price

def print_data(item, price):
    # Create and print a DataFrame
    data = {'Item': [item], 'Price': [price]}
    df = pd.DataFrame(data)
    print(df.to_string(index=False))

def process_data(item, price):
    item, price = cleanse_data(item, price)
    print_data(item, price)

# Example usage
item = "  Apple  "
price = "  1.50  "
item, price = cleanse_data(item, price)
print_data(item, price)
```

Use GitHub Copilot to rewrite conditional code for better readability

There are often several ways to write code that does, or does not, get executed depending on various conditions. Some conditional structures are better suited than others to particular use cases, and choosing an alternative conditional structure can sometimes make the code easier to read.

This Java method uses a series of `if` and `else if` statements to determine which operation to perform:

Java

```java
public string getSound(String animal) {
    if (animal == null) {
        System.out.println("Animal is null");
    } else if (animal.equalsIgnoreCase("Dog")) {
```

```
        return "bark";
    } else if (animal.equalsIgnoreCase("Cat")) {
        return "meow";
    } else if (animal.equalsIgnoreCase("Bird")) {
        return "tweet";
    }
    return "unknown";
}
```

A switch statement might be a better way of applying the same logic.

Java

```
/**
    * Returns the sound made by the specified animal.
    *
    * @param animal the name of the animal
    * @return the sound made by the animal, or "unknown" if the animal is
not recognized
    */
public String getAnimalSound(String animal) {
    return switch (animal) {
        case null -> "Animal is null";
        case "Dog" -> "bark";
        case "Cat" -> "meow";
        case "Bird" -> "tweet";
        default -> "unknown";
    };
}
```

Use GitHub Copilot to reformat code to use a different structure

Suppose you have the following function in JavaScript:

JavaScript

```
function listRepos(o, p) {
    return fetch(`https://api.github.com/orgs/${o}/repos?per_page=${parse-
Int()}`)
        .then(response => response.json())
        .then( (data) => data);
}
```

If your coding standards require you to use the arrow notation for functions, and descriptive names for parameters, you can use Copilot to help you make these changes.

JavaScript

```
const listRepositories = (organization, perPage) => {
    return fetch(`https://api.github.com/orgs/${organization}/repos?per_-
page=${parseInt(perPage)}`)
        .then(response => response.json())
        .then(data => data);
};
```

Summary

GitHub Copilot can help you to refactor code in various ways. You can use the Chat view or Inline Chat to ask Copilot to optimize inefficient code, clean up repeated code, make code more concise, split up complex units of code, rewrite conditional code for better readability, and reformat code to use a different structure.

Next unit: Refactor code using GitHub Copilot Inline Chat

Previous Next