GitHub Copilot

# Modernizing COBOL with GitHub Copilot

Mashrur Hossain, Enterprise Senior Solutions Engineer, GitHub

WRITTEN BY GITHUB WITH ♥

# Table of contents

# Introduction

GitHub is where over 100 million developers shape the future of software, together. The platform empowers developers to contribute to the open source community, build proprietary software, secure code, ship it anywhere, and leverage generative AI to accelerate software development.

In July of 2022, GitHub made GitHub Copilot generally available: this tool is an AI pair programmer that assists developers in building performant software more quickly while optimizing for their happiness. Below, we'll discuss how generative AI tools, like GitHub Copilot, are supercharging dev teams and making collaborative practices like innersource even more effective.

Legacy codebases, like those written in COBOL, serve as the backbone of many established industries, including finance, manufacturing, healthcare, and telecommunications. While these systems are sturdy and reliable, they are often bound to outdated paradigms that challenge integration with newer technologies, lead to increased maintenance costs, and face a diminishing talent pool for support. Migrating from COBOL to a more modern language is not just a matter of translation; it's a complete transformation, which can be overwhelming and filled with uncertainties.

Despite the need to evolve, many organizations hesitate, fearing the scale of the challenge and the potential for unforeseen disruptions. There's a vast difference between procedural COBOL and the contemporary, often object-oriented or event-driven, paradigms of languages like Java and Go. Manually navigating this migration journey can be arduous and error-prone. However, there's light at the end of this tunnel.

With advanced generative models like those which power GitHub Copilot Business, modernizing COBOL code can be more streamlined and precise. The Copilot Chat feature of GitHub Copilot can assist in translating code and offer insights into optimizing structures, integrating patterns, and preserving original functionalities through automated test generation. Through a series of steps, GitHub Copilot will serve to help your COBOL transformation journey to modern software. Anecdotally, we've seen organizations accelerate their migration journey by a double digit percentage by leveraging Copilot.

In the next sections, we'll share the best practices we've learned while helping top organizations prepare for an easier migration, harnessing the power of Copilot to breathe new life into age-old systems.

# The Challenges of Modernizing Legacy Code

To effectively transition from legacy systems, such as those built on COBOL, two critical developer processes must be honed and intertwined: **legacy code understanding** and **solution integration.** Developers should be able to swiftly grasp the existing legacy code, and simultaneously integrate modernized solutions into the evolving infrastructure.

## Challenges with Legacy Code Understanding

- **Bottlenecks:** Institutional knowledge of COBOL is fading. Many developers are more familiar with the latest languages and frameworks than those used to bring the enterprise online in the 20th century.
- **Interpreting legacy code:** Developers must often start by deciphering often decades-old, sprawling, undocumented, or poorly documented COBOL programs to understand their intent. This exhaustive process pushes developers to either cling to older systems longer or opt for total rewrites without complete understanding.
- **Previous attempts:** Simplifying the procedure using rudimentary translation tools or maintaining comprehensive documentation has largely been insufficient. With archaic coding conventions, monolithic architectures, and obsolete business logic embedded into the code, navigating a legacy system can feel like venturing through a dense, unmapped forest.

## Challenges with Solution Integration

- **Delicate integration:** Integrating modern solutions into older systems requires ensuring that the modernized code does not cause disruptions, inefficiencies, or vulnerabilities. Extensive testing, patching, and sometimes rolling back changes altogether are necessary to ensure smooth operations.
- **Developer burden:** Maintainers or senior developers have the added burden of reviewing the integration, grasping its intent, and determining if it aligns with the broader goal of modernization. This usually isn't their only responsibility, making the process even more tedious and leading to delays in modernization projects.

Moving away from legacy systems such as COBOL presents myriad complications. The declining knowledge in older languages and the daunting task of unraveling extensive, ambiguous code turns understanding the systems into a significant barrier. Efforts made in the past to ease this shift have largely been insufficient. Blending modern solutions into these age-old frameworks demands an intricate fusion of the past and present, with an emphasis on ensuring both stability and security. These impediments not only decelerate the pace of modernization but may also dissuade teams from pursuing these essential transformations. Let's take a practical look at how GitHub Copilot can provide a smoother transition from legacy systems like COBOL to more contemporary platforms.

# GitHub Copilot to Modernize Legacy COBOL to Java

Generative AI, typified by models that provide data recommendations tailored to user requirements, is emerging as a game-changer in legacy code conversion. Leveraging a platform like GitHub Copilot Business, which is integrated into the software transformation process, developers can experience a smoother transition from legacy systems like COBOL to contemporary platforms like Java.

Built upon AI models developed by GitHub, OpenAI, and Microsoft, GitHub Copilot can help streamline the process of understanding and translating age-old codebases, especially by shedding light on the complexities of legacy systems.

Traditionally, developers tasked with interpreting legacy code turned to manuals, older documentation, or external resources to bridge knowledge gaps. GitHub Copilot's Chat feature serves as an in-context guide, offering insights directly within the editor. Feedback indicates that a significant portion of developers using GitHub Copilot report reduced time spent on external solution searches and experience a more uninterrupted workflow during legacy code modernization.

Copilot is trained on expansive datasets including diverse coding conventions, languages, and paradigms. The result is an AI model that can respond with relevant code suggestions in milliseconds. With the aid of this AI, developers working on COBOL-to-[insert language of choice] transitions have observed substantial gains in velocity and productivity.

Developers can highlight COBOL code and prompt Copilot Chat to explain or summarize what the code does. The developer can then prompt Copilot Chat to produce the conversion to their language of choice, then through a practice termed "comment-driven development", developers can detail the functionalities they aim to preserve or introduce. Beyond just reactive responses, Copilot can also actively review the ongoing work, proposing snippets or structures for subsequent lines or components.

Copilot's suggestions are not generic; they're tailored based on the active files in the IDE, ensuring that the offered solutions better align with the project's structure, style, and requirements. As the model matures, that contextual awareness will continue to improve, for example we previously transitioned Copilot Chat from an 8k token, 3.5 model to 32k GPT-4. The greater token volume translates to more capacity for the model to understand the prompts as more natural language characters are computed.

As legacy systems like COBOL continue to challenge industries, tools like GitHub Copilot become invaluable allies, accelerating the journey towards modern, efficient, and scalable infrastructures.

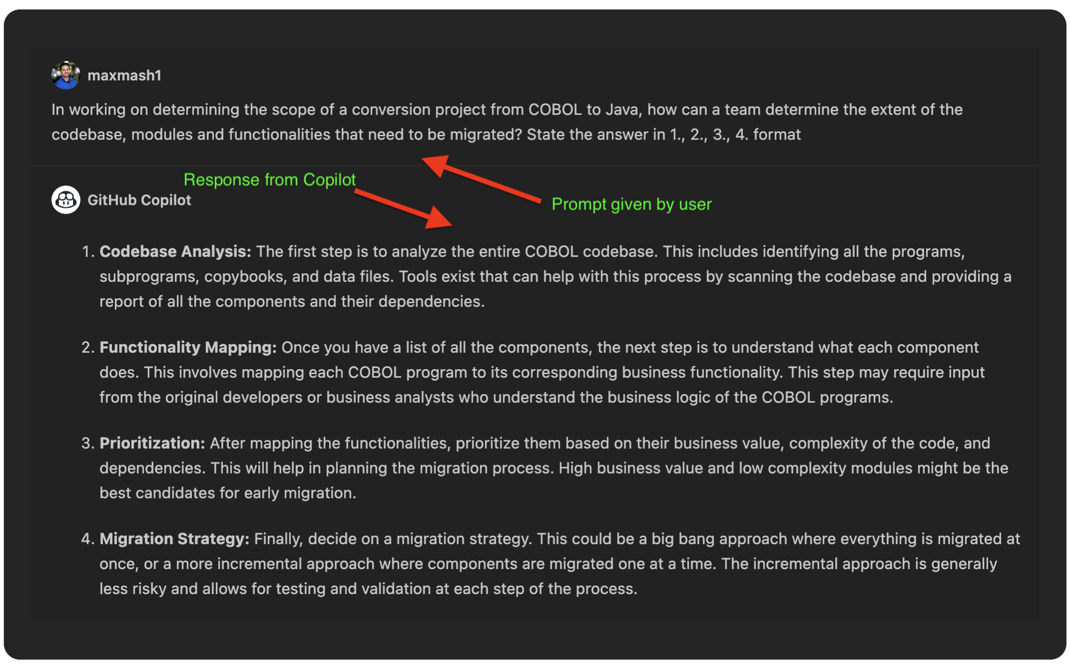
# Steps for Conversion Projects

## Purpose and scope

There are various approaches to planning a conversion project from COBOL to a modern language like Java. This involves more than just converting COBOL code due to the differences in paradigms, platforms, and ecosystems. In this chapter, we will explore several steps that can be considered for these projects. However, before diving into specific steps, it's crucial to first establish a high-level understanding of the project's purpose and scope.

- **Purpose of conversion:** Understand why the conversion is being undertaken. Whether it's for easier maintenance, integration with modern systems, cost-saving, talent availability, or other reasons.
- **Scope of conversion:** Determine the extent of the codebase, modules, and functionalities that need to be migrated.

A helpful initial step is to ask Copilot Chat 'how?'. You'll find that Copilot's response is fairly generic to start with, but we'll dive into more detailed responses shortly. Take a look:



This was a general prompt given to GitHub Copilot with no prior knowledge of the context for Copilot to base its response on. But as we can see, Copilot has done a pretty good job of identifying the baseline steps to determine the scope of the conversion project.

Now, let's explore each result in more detail by refining our prompt and/or asking follow-up questions based on the response. For instance:

> **maxmash1**
>
> Recommend in detail how a team should go about working on functionality mapping as outlined in your initial response under step 2. Also include where you (Copilot) can help in the process, for example, can you generate summaries or extract business logic from highlighted COBOL code in files?
>
> **GitHub Copilot**
>
> Functionality mapping is a crucial step in a COBOL to Java conversion project. Here's a detailed approach on how a team can go about it:
>
> 1. **Identify Key Components:** Start by identifying the key components of each COBOL program. This includes data divisions, file divisions, working-storage sections, procedure divisions, and any other significant sections. Understanding these components will give you a high-level overview of what each program does.
> 2. **Understand Data Flow:** Next, try to understand the data flow within each program. Look at where data is coming from (input files, databases, user input, etc.) and where it's going (output files, databases, etc.). This will give you a sense of what the program is processing.
> 3. **Analyze Procedure Division:** The Procedure Division is where the main business logic of a COBOL program resides. Analyze each paragraph and section to understand what operations are being performed. Look for any business rules or calculations that are being applied.
> 4. **Map to Business Functionality:** Once you understand what each program is doing, try to map it to its corresponding business functionality. This might require input from business analysts or subject matter experts who understand the business processes that the COBOL programs are supporting.
> 5. **Document Findings:** Document your findings in a way that's easy for the rest of the team to understand. This could be a spreadsheet or a diagram that shows each COBOL program and its corresponding business functionality.
>
> As GitHub Copilot, I can assist in several ways:
>
> - **Code Explanation:** I can help explain individual COBOL code snippets, including what different sections and statements are doing. This can help you understand the structure and functionality of the code.
> - **Business Logic Extraction:** While I can provide a general explanation of what the code is doing, extracting the specific business logic can be challenging, especially if the code is complex or lacks comments. However, I can certainly help you understand the operations being performed in the code, which can give you some insight into the business logic.
> - **Documentation:** I can assist in generating summaries or descriptions of the code, which can be included in your documentation. However, these summaries would be based on the code itself, and might not fully capture the business logic or business functionality.
>
> Remember, while I can provide valuable assistance, I'm a tool to aid your understanding and analysis, not a replacement for a thorough manual review by experienced developers or analysts.

Notice how the response from GitHub Copilot is tailored to the task that was outlined in the prompt. Copilot can do a wonderful job of translating COBOL code to Java (examples to follow in sections below). However, since that was not relevant to the task at hand, it did not include that as an item in its steps.

Several leading companies in the financial services and telecom industries have already embarked on modernization projects to transition away from COBOL. Let's look at two examples below: first, a project plan from a financial services company outlining their vision for the modernization project, and second, a telecommunications company's reasoning, based on a thorough assessment of tools, for choosing GitHub Copilot for their project.

# Example project plan

An example project plan of a COBOL to Java conversion project using GitHub Copilot is detailed below. We split it up in two phases:

**Prerequisites:**

- Measure current velocity.
- Select applications to focus on for migrations based on priority, business impact, readiness.
- Prepare the development environment so code is accessible to Copilot.

**Transforming software:**

- **Stage 1:** Get an explanation for COBOL code.
- **Stage 2:** Generate new code that performs the function. Alternatively, translate the COBOL code directly where necessary.
- **Stage 3:** Generate test cases based on the program in modernized code. Does the newly generated code still do what the customer expects?
- **Stage 4:** Measure Performance and Accuracy.
- **Stage 5:** Develop Supplemental Models or implement RAG (Retrieval Augmented Generation) for specific use cases.

These stages are perfectly described as high level implementation steps from a customer who deals with large scale projects and it closely resembles the outline we received from Copilot itself. Let's draw attention to stage 5: Supplemental model for specific use case - this is where an LLM is trained on customer code so the AI can inherently understand and generate responses aligned with specific coding practices and domain-specific knowledge unique to the customer and further accelerate their development. Having this as stage 5, after measurement, results in the customer being able to quantify the acceleration in code generation, bug fixes, documentation and more.

**Another example** - a leading telecom company decided to take on their COBOL modernization project using GitHub Copilot after a thorough assessment of the tool and alternatives. They tasked roughly a dozen engineers with evaluating Copilot and other tools over the course of a month. The developers used Copilot and other tools in real-world development – specifically, migrating COBOL to Java. At the end of this evaluation, they created a scorecard comparing the tools.

One of the methods used was to leverage APIs for the latest GPT models in a simple chat-based web application they built internally. Developers had to manually reconstruct their prompt for each query. Overall, they gave this process high marks for the model responses.

- Suggestions were of high quality.
- They could translate much larger "chunks" of code in a single pass (due to the larger token budget).

However, they opted to leverage GitHub Copilot instead for the following reasons:

- Seamless integration with the IDE provided a smoother experience.
- No need to manually create prompts, as Copilot offers advanced server-side prompt engineering out of the box.
- Copilot utilizes the latest GPT model.

**Now that we have seen the “how’s and why’s” of industry titans choosing Copilot for their COBOL modernization projects, let’s dig into some of the details of what such a project may look like.**

# Process

## Initial assessment

Complexity analysis: Understand the size and complexity of the COBOL codebase. GitHub Copilot Chat can play a pivotal role in this step, specifically for:

a. **Code explanation:** Copilot Chat can help explain individual COBOL code snippets, including what different sections and statements are doing. This can help a team understand the structure and functionality of the code, which is a crucial part of complexity analysis.

b. **Identifying code patterns:** Copilot Chat can help identify common code patterns, such as loops, conditional statements and procedure calls. Understanding these patterns can give a sense of the complexity of the code.

c. **Understanding data structures:** Copilot Chat can assist in understanding the data structures used in the COBOL code, such as arrays, records, and files. This can help in assessing the complexity of the data manipulation in the code.

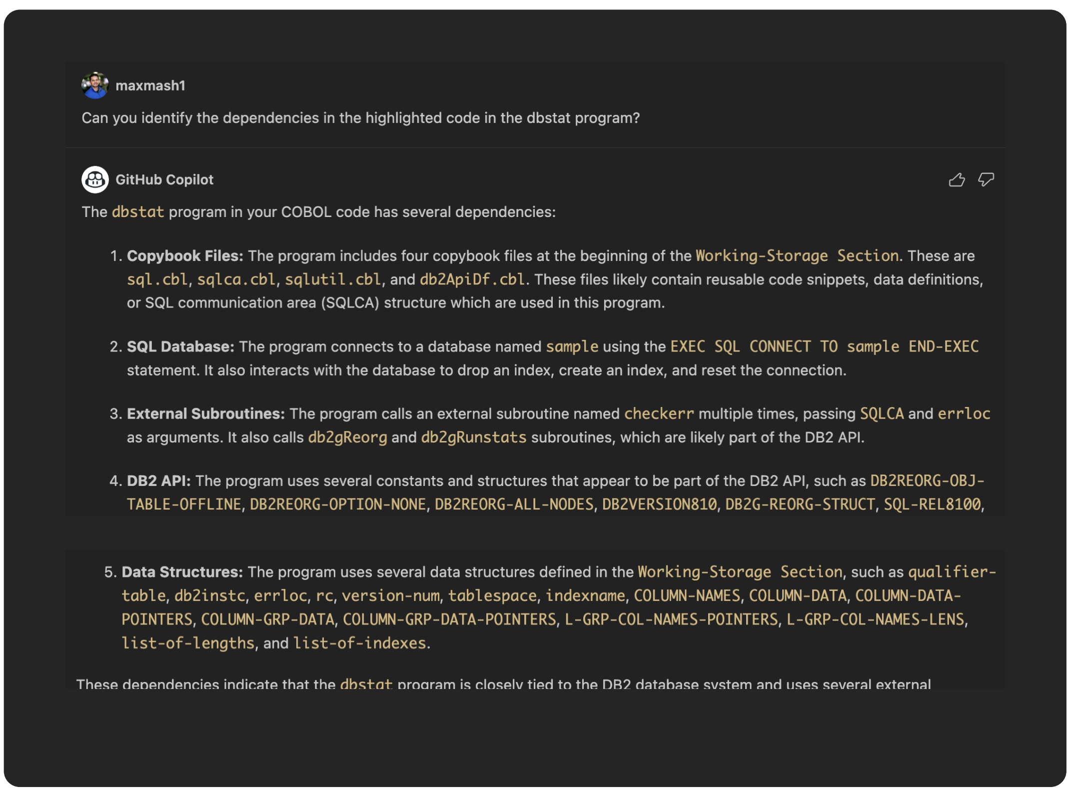
d. **Summary and comments:** Copilot Chat can be used to generate/add comments to code snippets including summarizing entire sections (sometimes hundreds of lines of code with 1 prompt) to better understand what each part of COBOL programs are doing.

e. **Generate documentation:** Copilot Chat can be prompted to generate documentation in plain text from programs by simply highlighting sections and prompting it to generate documentation in the format that the user wants.

f. **Dependencies:** Identify external systems, databases, and services the COBOL application interacts with.

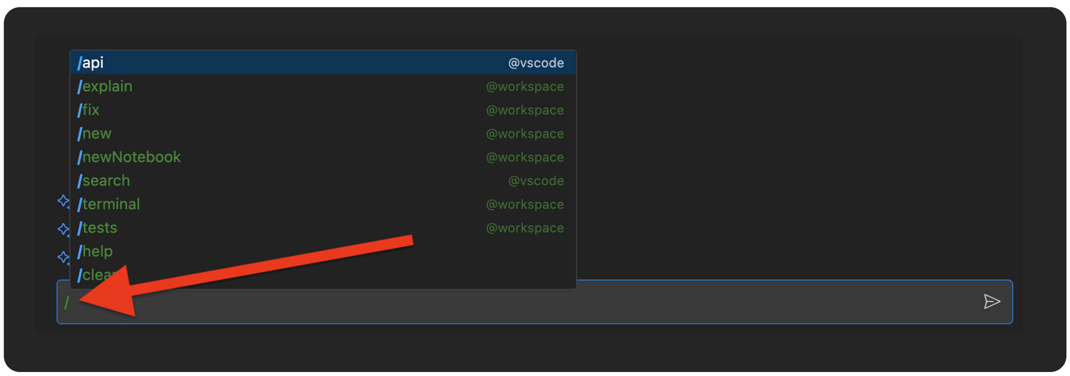
While GitHub Copilot can't scan multiple files at once (like an SCA tool can, at least at the time of writing this whitepaper), it can be prompted to identify dependencies within a single COBOL program. It will also provide relevant information on the dependencies and what needs to be done with them in follow-up steps, providing a clear sense of the codebase's complexity.

Let's look at an example. We copied the code (without the comments or summary) from a dbstat.sqb file provided by IBM in its official documentation page for Db2 here. Then, we asked Copilot to identify the dependencies. Images of the interaction can be found below:

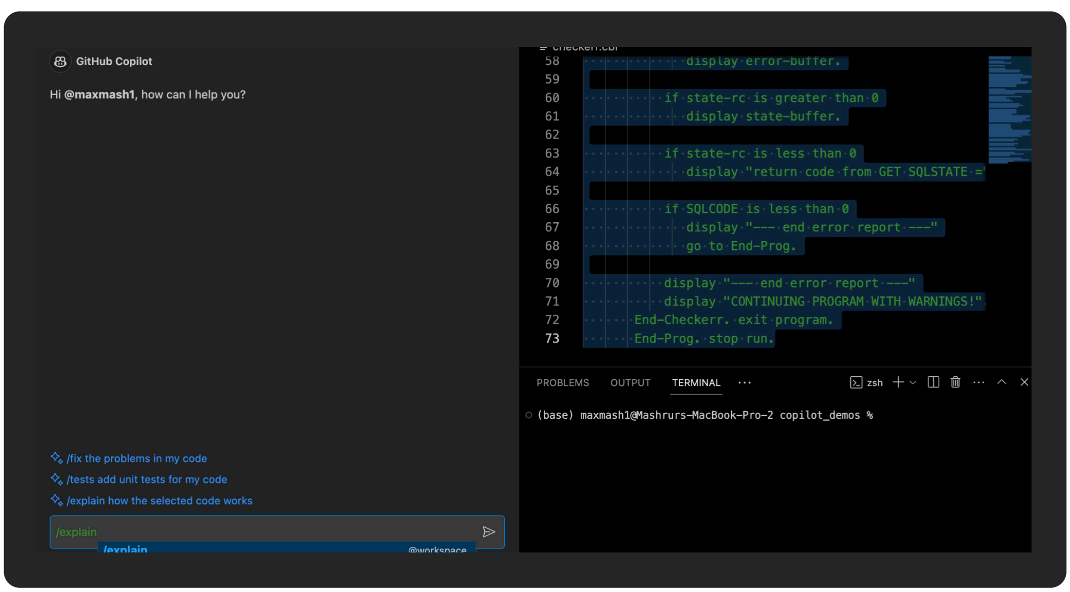


As we can see from the response above, Copilot does a splendid job of not only identifying the dependencies, but also providing information on what they might be used for. This information can be extremely valuable for a migration team.

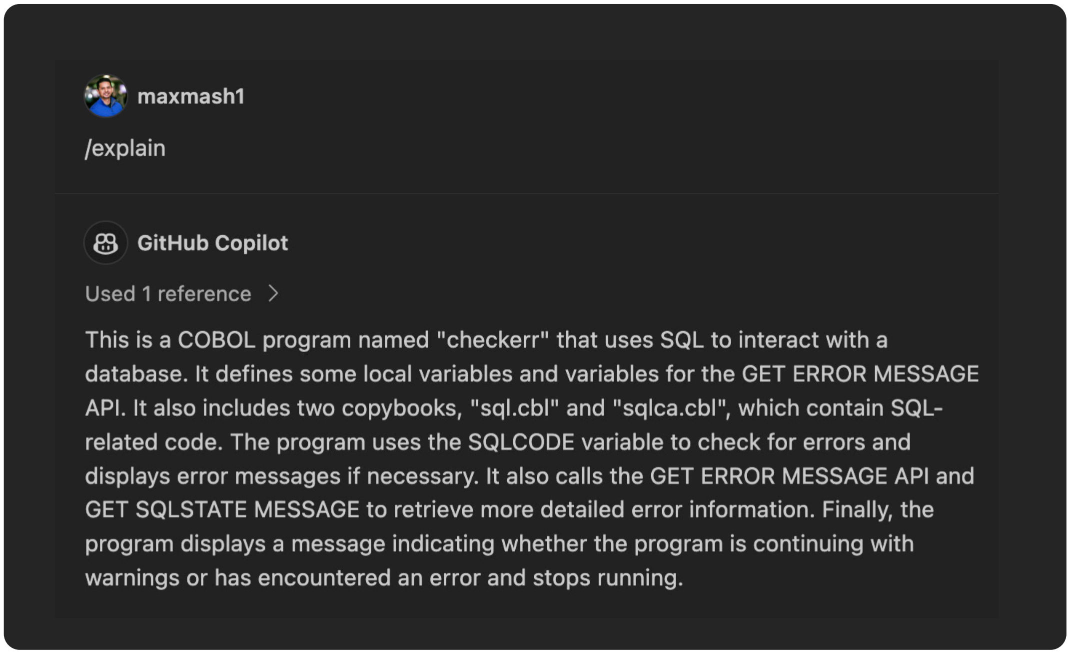
Now let's try a slash command, specifically "/explain" and have Copilot explain the code block to us. The available slash commands can be pulled up by entering a slash in the prompt window.



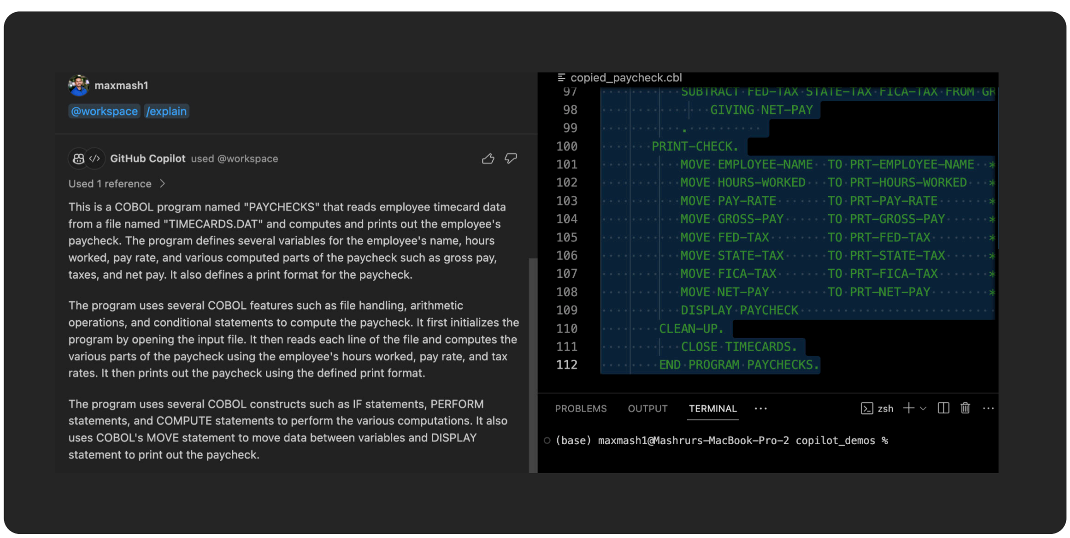

To use the "/explain" command, highlight all the code you want explained then enter the command as shown below:



Let's run it to see the output:




A similar approach can be used to identify subroutines, we will see what the response will be when identifying subroutines from a payroll program that deals with employee payroll data when the subroutines were referenced but not defined in the highlighted code. But first, let's try "/explain" to get a thorough understanding of what the program does:



Now let's prompt Copilot to identify and provide details about the subroutines:

> **maxmash1**
>
> Identify the subroutines in the highlighted code
>
> **GitHub Copilot**
>
> In the highlighted COBOL code, there are two subroutines:
>
> 1. `FEDERAL_TAX_SUBROUTINE`: This subroutine is called in the `X000-CALC-FED-TAX` section. It takes two arguments: `WS-GROSS-PAY` and `WS-FED-TAX`.
>
> 2. `STATE_TAX_SUBROUTINE`: This subroutine is called in the `X100-CALC-STATE-TAX` section. It takes two arguments: `WS-GROSS-PAY` and `WS-STATE-TAX`.
>
> These subroutines are likely defined in other parts of the code or in separate COBOL programs. They are used to calculate the federal and state taxes based on the gross pay.

As observed from the response above, Copilot is able to identify and provide high level details of the subroutines from the highlighted code.

## Choose a Target Language

A commonly chosen language is Java. Java is suitable for large-scale, enterprise applications that need robustness, security, and performance. It also integrates well with other enterprise systems.

## Code Translation

**Automated translation with Copilot:** Copilot can be used to automatically convert COBOL code to a target language like Java or Go. It can also be used to generate extensive unit tests and ensure the resulting code is providing the output that is desired. Below is an example of when a translation is attempted from a COBOL file taken from the previously referenced IBM DB2 examples here and what the potential response could look like. The COBOL program checks for and prints SQL warnings and errors to the screen:

maxmash1

can you convert the highlighted code to java?

GitHub Copilot

Sure, here's a rough translation of the COBOL code to Java. Note that this is a simplified version and doesn't include file handling or formatting output exactly as in the COBOL program.

```java
import java.sql.*;

public class CheckError {
    private static final int BUFFER_SIZE = 1024;
    private static final int LINE_WIDTH = 80;
    private char[] errorBuffer = new char[BUFFER_SIZE];
    private char[] stateBuffer = new char[BUFFER_SIZE];

    public void checkError(SQLException sqlException, String errloc) {
        if (sqlException.getErrorCode() == 0) {
            return;
        }

        System.out.println("--- error report ---");
        System.out.println("ERROR occurred : " + errloc);
        System.out.println("SQLCODE : " + sqlException.getErrorCode());

        // Get error message
        int errorRc = getErrorMessage(sqlException, errorBuffer);
        if (errorRc > 0) {
            System.out.println(new String(errorBuffer).trim());
        }

        // Get SQL state message
        int stateRc = getSqlStateMessage(sqlException, stateBuffer);
        if (stateRc > 0) {
            System.out.println(new String(stateBuffer).trim());
        } else if (stateRc < 0) {
            System.out.println("return code from GET SQLSTATE = " + stateRc);
        }

        if (sqlException.getErrorCode() < 0) {
            System.out.println("--- end error report ---");
            System.exit(0);
        }

        System.out.println("--- end error report ---");
        System.out.println("CONTINUING PROGRAM WITH WARNINGS!");
    }

    private int getErrorMessage(SQLException sqlException, char[] errorBuffer) {
        // This is a placeholder. Replace with actual implementation.
        return 0;
    }

    private int getSqlStateMessage(SQLException sqlException, char[] stateBuffer) {
        // This is a placeholder. Replace with actual implementation.
        return 0;
    }
}
```

This Java code creates a `CheckError` class with a `checkError` method that takes a `SQLException` and a string `errloc` as parameters. The `checkError` method prints an error report if the SQL error code is not 0. It calls `getErrorMessage` and `getSqlStateMessage` methods to get the error message and SQL state message, respectively.

Next steps:

- Implement the `getErrorMessage` and `getSqlStateMessage` methods.
- Call the `checkError` method when a `SQLException` is caught.

As observed in the output above, Copilot not only attempts to translate the code to Java but also provides useful comments and suggests next steps (e.g., placeholder comments for where SQL code will be needed).

Another area in translation where Copilot excels, is when there is either not enough information or code is available for a complete translation, or when a direct translation to the target language is not directly possible due to use of custom code, patterns, libraries that are specific to the legacy language (COBOL in this case).

Below we have an example of what the output looks like if a direct translation is attempted in such a case using the previously referenced [dbstat.sqb program](dbstat.sqb program) (the utility program can be used to reorganize tables and run statistics), Copilot provides a lot of comments and steps in addition to the code in the target language to help the team accomplish the task at hand:

Here's a simplified Java version of the COBOL code. Note that this doesn't include all the details of the original COBOL program, such as file handling or formatting output.

```java
import java.sql.*;

public class DBStat {
    private String qualifierTable;
    private String statement;
    private String db2instc;
    private String errloc;
    private int rc;
    private int versionNum;
    private int tablespaceLen;
    private String tablespace;
    private int indexnameLen;
    private String indexname;
    private int qualifierTableLen;
    private int numIndexes;
    private String[] columnNames = new String[5];
    private String[] columnData = new String[2];
    private String[] columnDataPointers = new String[2];
    private String[] columnGrpData = new String[2];
    private String[] columnGrpDataPointers = new String[2];
    private String[] grpColNamesPointers = new String[2];
    private int[] grpColNamesLen = new int[5];
    private int iLength;
    private String lIndex;

 public void dbstat() {
     System.out.println("Sample Java program: DBSTAT");

     // Connect to database
     try (Connection conn = DriverManager.getConnection("jdbc:db2://localhost:50000/sample", "db2instc", "")
         Statement stmt = conn.createStatement();

         // Drop index
         stmt.execute("DROP INDEX sample.testind");

         // Create index
         stmt.execute("CREATE INDEX sample.testind ON staff (salary)");

         // Reorganize table
         // This is a placeholder. Replace with actual implementation.

         // Run statistics
         // This is a placeholder. Replace with actual implementation.

         // Reset connection
         // This is a placeholder. Replace with actual implementation.

     } catch (SQLException e) {
         checkError(e, errloc);
     }
 }


    private void checkError(SQLException sqlException, String errloc) {
        // This is a placeholder. Replace with actual implementation.
    }
}
```

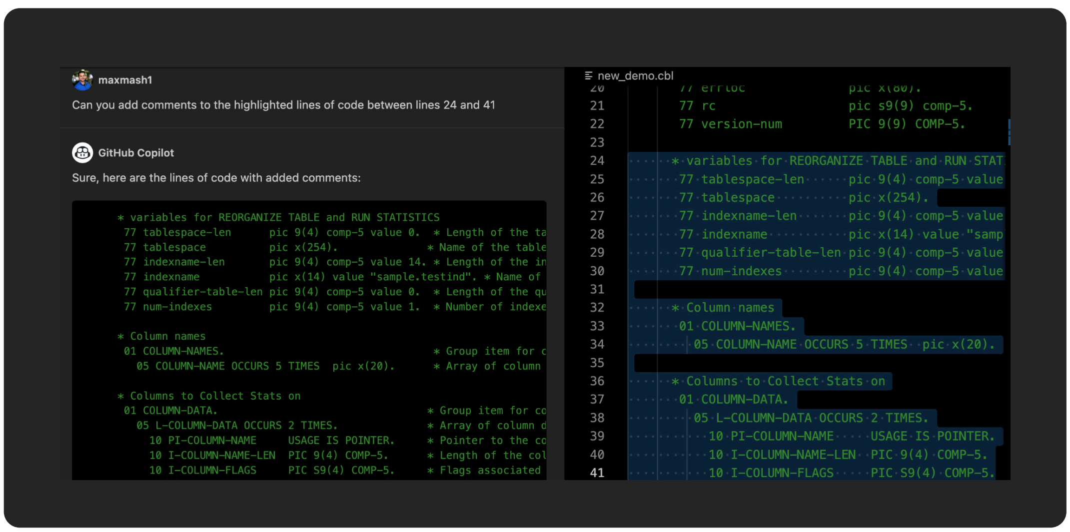
This Java code creates a `DBStat` class with a `dbstat` method that connects to a DB2 database, drops an index, creates a new index, reorganizes a table, and runs statistics. The `checkError` method is called when a `SQLException` is caught.

Next steps:

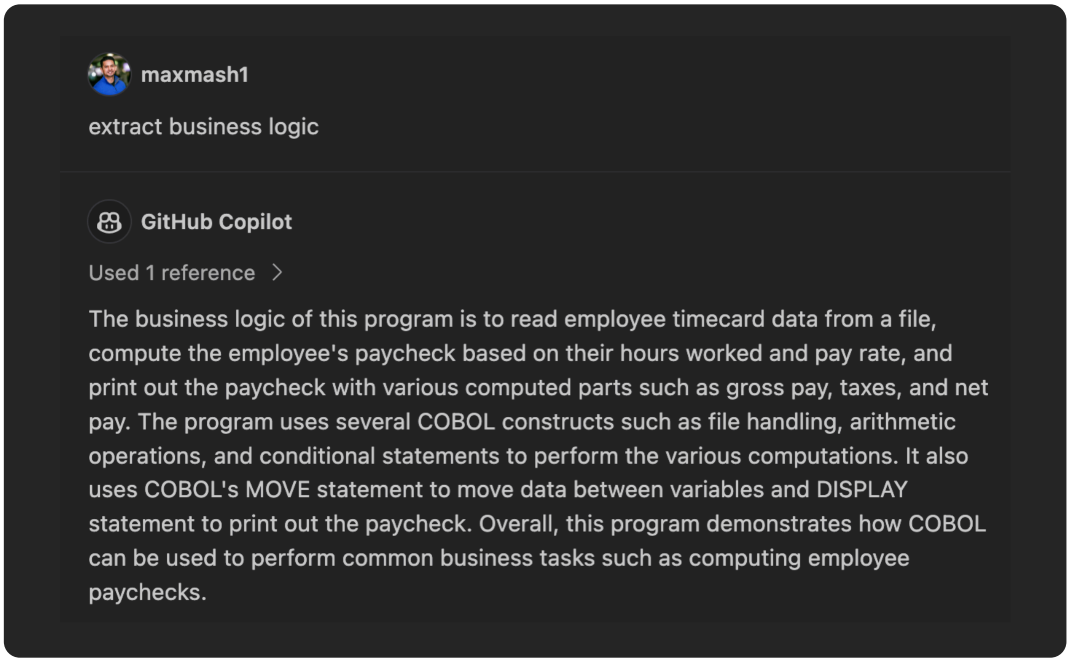
- Implement the `checkError` method.
- Implement the table reorganization and statistics running.
- Replace the placeholders with actual implementation.

- **Manual rewrite using Copilot:** This can be a very strong approach when the intent is to not just translate but to modernize the applications and infrastructure as a whole. A manual rewrite can help ensure that the application takes full advantage of the features and strengths of the new language and associated frameworks. This is also an opportunity to refactor and improve the design since chunks of code can be highlighted and refactored with the click of a button. This can be used in conjunction with step 1 above to enhance the new application and its functionality.

- **Code explanation:** Copilot can be used to understand COBOL code when considering translation, redesign, optimization. Using Copilot Chat, blocks of code can be highlighted and prompted for various questions like explanation, summary and comment generation, refactoring and much more. We already saw the "/explain" command used to generate explanations of code above. Below is an example where Copilot is asked to add comments to a block of code:



Below is an example of extracting the business logic from the previously referenced payroll program:




## Database migration

COBOL often interacts with legacy databases (like IBM DB2): Transitioning to modern databases can also be considered with the aid of GitHub Copilot in addition to tools or services that can assist with data migration. Some examples of steps that can be taken at each stage of the database migration process, and the role Copilot can play in them, are described below (MySQL and PostgreSQL examples used for modern databases as examples, can be applied to any):

**Data mapping:**

- Utilize Copilot to generate SQL scripts for creating table structures in the target database by providing desired schema changes and transformation requirements.
- Prompt Example: “Create a MySQL table definition equivalent to this DB2 table definition ....”

**Optimization Tips:**

- Generate indexing and optimization strategies by inputting the table structures and query patterns into Copilot.
- Prompt example: "Suggest indices for optimizing these SQL queries on a PostgreSQL table: ...."

**Testing strategies:**

- Copilot can suggest various testing strategies and generate SQL queries and scripts to validate post-migration data.
- Prompt example: "Suggest a strategy and queries to validate the data after migrating from DB2 to PostgreSQL."

**Rollback procedures:**

- Develop a rollback plan with the aid of Copilot, including crafting scripts to revert changes or restore data from backups.
- Prompt example: "Create a script to restore data from a backup file to the PostgreSQL table in case of migration failure."

**Documentation assistance:**

- Copilot can help generate content for documenting schema, relationships, transformations, and the migration process, aiding future reference and knowledge sharing.
- Prompt example: "Help me create documentation for the migration process, including changes in schema, data types, and transformations made from DB2 to MySQL."

**Post-migration support:**

- Copilot can suggest monitoring and performance tuning strategies, as well as help troubleshoot issues with the new database system.
- Prompt example: "Provide a strategy for monitoring and optimizing the performance of this PostgreSQL table after migration."

## Testing

Unit Testing: Given the drastic change in the codebase, it is very important to ensure all functions are covered by unit tests. Copilot thrives in generating extensive unit tests for blocks of code, functions, classes and sometimes entire files using just a prompt. There are useful shortcuts available like /tests to speed up the process as well.

Below is a result of running the "/tests" command on the translated Java code from the payroll program we were working with earlier:

maxmash1

@workspace /tests

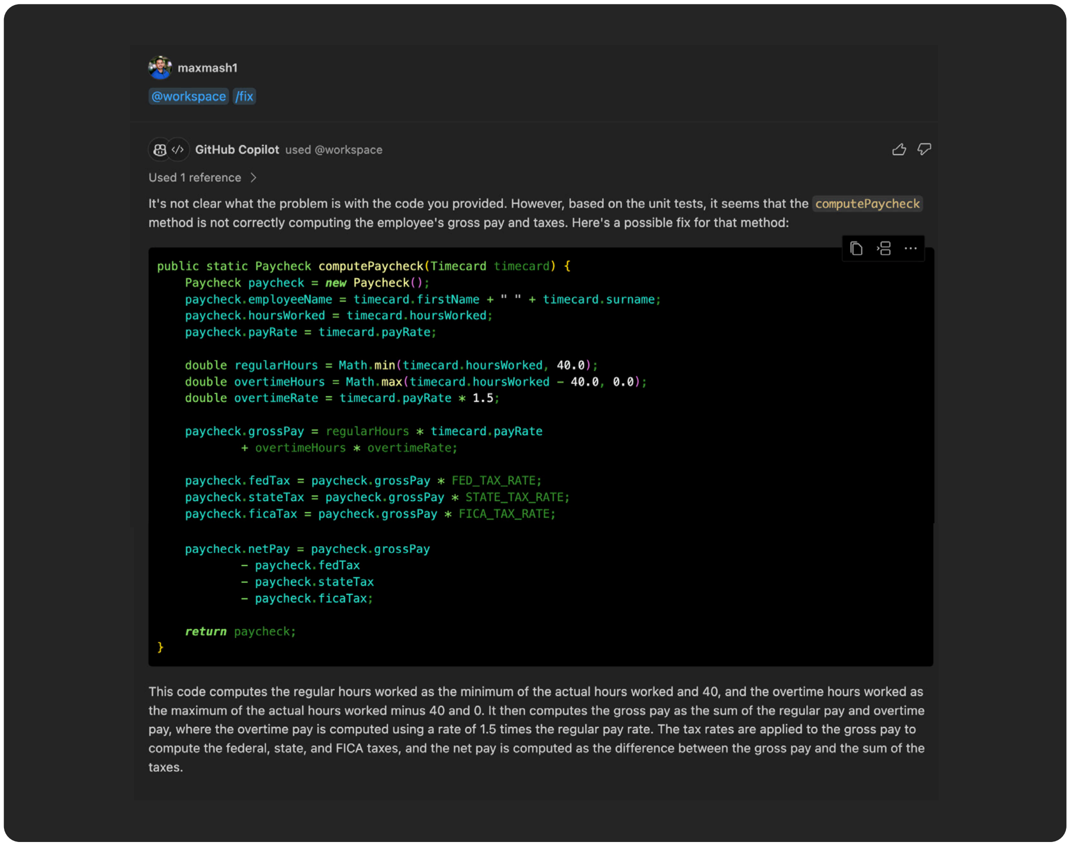
GitHub Copilot used @workspace

Sure, here's an example set of unit test functions for the Paychecks class:

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringWriter;

public class PaychecksTest {
    @Test
    public void testParseTimecard() {
        String line = "John Smith 401234";
        Paychecks.Timecard timecard = Paychecks.parseTimecard(line);
        assertEquals("John", timecard.firstName);
        assertEquals("Smith", timecard.surname);
        assertEquals(40.1, timecard.hoursWorked);
        assertEquals(234.0, timecard.payRate);
    }
    @Test
    public void testComputePaycheck() {
        Paychecks.Timecard timecard = new Paychecks.Timecard();
        timecard.firstName = "John";
        timecard.surname = "Smith";
        timecard.hoursWorked = 45.0;
        timecard.payRate = 200.0;

        Paychecks.Paycheck paycheck = Paychecks.computePaycheck(timecard);
        assertEquals("John Smith", paycheck.employeeName);
        assertEquals(45.0, paycheck.hoursWorked);
        assertEquals(200.0, paycheck.payRate);
        assertEquals(10000.0, paycheck.grossPay);
        assertEquals(1640.0, paycheck.fedTax);
        assertEquals(700.0, paycheck.stateTax);
        assertEquals(620.0, paycheck.ficaTax);
        assertEquals(7040.0, paycheck.netPay);
    }

    @Test
    public void testWritePaycheck() throws IOException {
        Paychecks.Paycheck paycheck = new Paychecks.Paycheck();
        paycheck.employeeName = "John Smith";
        paycheck.hoursWorked = 45.0;
        paycheck.payRate = 200.0;
        paycheck.grossPay = 10000.0;
        paycheck.fedTax = 1640.0;
        paycheck.stateTax = 700.0;
        paycheck.ficaTax = 620.0;
        paycheck.netPay = 7040.0;

        StringWriter sw = new StringWriter();
        PrintWriter output = new PrintWriter(sw);
        Paychecks.writePaycheck(output, paycheck);
        output.close();

        String expected = "John Smith                45.0 200.00 $10,000.00 $1,640.00 $700.00 $620.00 $7,040
        assertEquals(expected, sw.toString());
    }

    @Test
    public void testMain() throws IOException {
        // Create input file
        String inputFileName = "test_input.txt";
        PrintWriter input = new PrintWriter(inputFileName);
        input.println("John Smith 401234");
        input.println("Jane Doe 402345");
        input.close();

        // Run program with test input file
        String outputFileName = "test_output.txt";
        Paychecks.main(new String[] {inputFileName, outputFileName});

        // Read output file and check contents
        BufferedReader output = new BufferedReader(new FileReader(outputFileName));
        assertEquals("John Smith                40.1 234.00 $9,374.00 $1,536.56 $655.18 $584.94 $6,596.32",
        assertEquals("Jane Doe                  40.0 234.00 $9,360.00 $1,531.04 $652.80 $582.48 $6,593.68",
        assertNull(output.readLine());
        output.close();
    }
}
```

To help verify the accuracy of the generated tests, we can prompt Copilot to either "fix" the code by using the "/fix" shortcut or we can write our prompt.

Below is an example of the output received when using the "/fix" command on the generated unit tests:




It is always a good idea to double-check our work and Copilot provides an easy to use "out of the box" solution to help us do just that.

- **Integration testing:** Do the new functions/modules integrate well with each other? This is crucial for ensuring the new application interfaces correctly with other systems, especially if the original COBOL system had numerous integrations. Copilot can be used extensively to generate integration tests for converted modules in applications.
- **User Acceptance Testing (UAT):** Help ensure end-users or stakeholders validate the functionality of the converted system. Copilot

can be used to generate scripts for users to follow as they test the system.

## Training & Documentation

Both development teams and users will need training since they're moving from a procedural paradigm (COBOL) to an object-oriented/event-driven paradigm like Java. Extensive documentation needs to be created and Copilot can help with document generation.

- Update existing documentation to reflect changes, architectures, and new workflows.

## Rollout strategy

A phased approach could be considered, rolling out modules or functionalities gradually.

- Thorough monitoring strategies for the new system after rollout looking for issues and performance bottlenecks.

## Post-migration maintenance

Establish a team or process to maintain, update, and troubleshoot the new codebase.

- Continue to gather feedback from users and stakeholders to improve the system post-migration.

## Considerations & challenges

- **Performance:** Operations that performed efficiently in COBOL might not do so in Java. Profiling and optimization may be required, and Copilot assists in optimizing various aspects of the newly developed application.
- **Legacy System Knowledge:** It's vital to involve experts in both COBOL and the target language (Java) during this transition to ensure that all nuances and functionalities are accurately transferred.
- **Integration with Legacy Systems:** If the entire conversion is not happening simultaneously, the new system may need to coexist and integrate with the existing COBOL systems for some time.

# Conclusion

Modernizing COBOL applications is a complex but necessary effort for organizations seeking to remain competitive in today's technology landscape. GitHub Copilot offers a powerful solution to streamline this process by providing contextual assistance, code translation, and documentation generation. By leveraging its capabilities, developers can significantly reduce the time and effort required to understand and convert legacy code, ultimately leading to more efficient migration projects. The structured approach outlined in the paper, including initial assessments, code translation, and thorough testing, ensures that organizations can effectively manage the challenges associated with legacy systems. As demonstrated by industry leaders, the integration of GitHub Copilot not only enhances productivity but also mitigates risks associated with modernization.

Embracing these tools and methodologies will enable organizations to transform their COBOL systems into modern, scalable solutions that meet current and future demands.

Want to learn more? Visit https://github.com/features/copilot
or chat with a member of our sales team.

WRITTEN BY GITHUB WITH ❤