Lab Assignment Report 09

CS202: Software Tools and Techniques for CSE

Roll Number: 22110087
Name: Parth Govale

Contents

1 Introduction, Setup, and Tools **2**
- 1.1 Overview . . . 2
- 1.2 Environment Setup . . . 2

2 Methodology and Execution **2**
- 2.1 Software Tool Setup . . . 2
- 2.2 Repository Selection and Criteria . . . 2
- 2.3 Dependency Analysis Using `pydeps` (Python) . . . 4
- 2.4 Dependency Impact Assessment . . . 6
- 2.5 Cohesion Analysis Using LCOM (Java) . . . 6
- 2.6 Analysis of High LCOM Values in Classes . . . 7
  - 2.6.1 What does a high LCOM value suggest about a class's design? . . . 7
  - 2.6.2 Is there a chance for performing functional decomposition? . . . 7

3 Results and Analysis **8**
- 3.1 Dependency Analysis (Python) . . . 8
- 3.2 Cohesion Analysis (Java) . . . 8
- 3.3 Comparative Table of LCOM Metrics . . . 9

4 Discussion and Conclusion **9**
- 4.1 Challenges and Reflections . . . 9
- 4.2 Lessons Learned . . . 9
- 4.3 Summary . . . 10

5 Appendix **10**
- 5.1 Commands Used . . . 10
- 5.2 Additional Resources . . . 12

1 Introduction, Setup, and Tools

1.1 Overview

This lab focuses on analyzing module dependency and class cohesion for both Python and Java projects. In the Python project, the `pydeps` tool is used to generate dependency graphs, calculate fan-in and fan-out metrics, and identify coupling issues. In the Java project, Lack of Cohesion of Methods (LCOM) metrics (LCOM1–LCOM5 and YALCOM) are used to evaluate class cohesion, detect design flaws, and identify opportunities for refactoring.

1.2 Environment Setup

- **Operating System:** Ubuntu 20.04 (Windows 11 VM SET-IITGN-VM)
- **Python Version:** 3.10.11
- **Java Version:** openjdk 17.0.14
- **Required Tool:** `pydeps` (v3.0.1)

2 Methodology and Execution

2.1 Software Tool Setup

- Set up the virtual machine using the provided `SET-IITGN-VM.ova` VM file.
  1. Downloaded the '.ova' file from the provided link: SET-IITGN-VM.
  2. Imported the '.ova' file into VirtualBox:

     ```
     File > Import Appliance > Select .ova file > Continue > Import
     ```

  3. Successfully launched the virtual machine.

2.2 Repository Selection and Criteria

1. Used the SEART GitHub Search Engine to identify suitable repositories.
2. Selection criteria included:
   - Language: Python
   - Number of commits: $\geq 4,000$
   - Number of stars: $\geq 5,000$
   - Number of forks: $\geq 1,000$

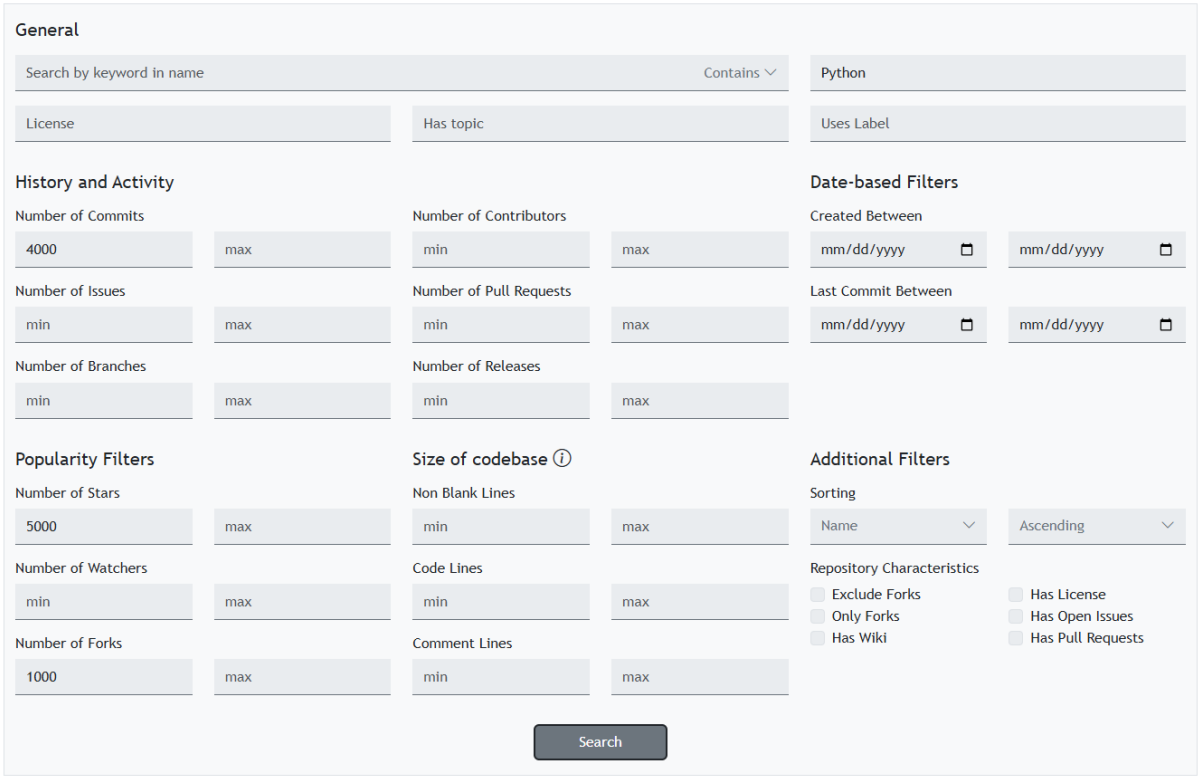


Figure 1: Search parameters in SEART Github Search Engine

3. Selected Python Repository: `fastapi` (GitHub: `https://github.com/fastapi/fastapi`).

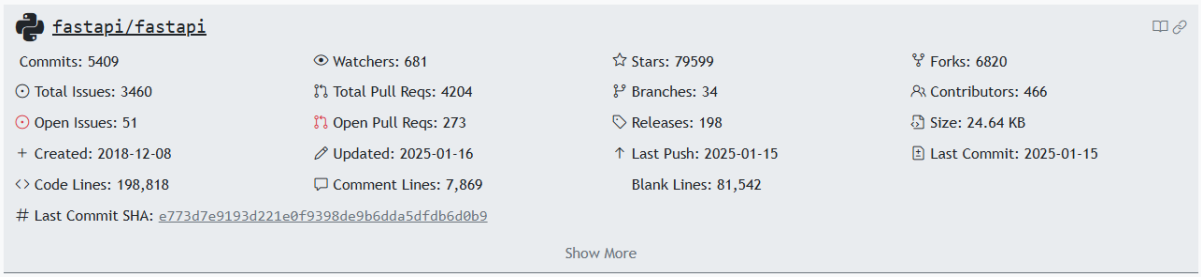


Figure 2: Fastapi GitHub Repository

4. Selected Java Repository: `jenkins` (GitHub: `https://github.com/jenkinsci/jenkins`).

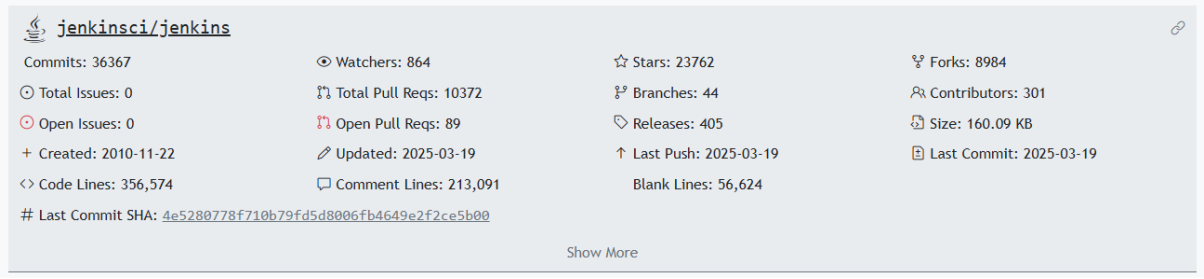


Figure 3: Jenkins GitHub Repository

2.3 Dependency Analysis Using `pydeps` (Python)

- Running `pydeps` with the command:

```
pydeps . --show-deps
```

- Dependency graph which was further analyzed to compute fan-in and fan-out for each module.

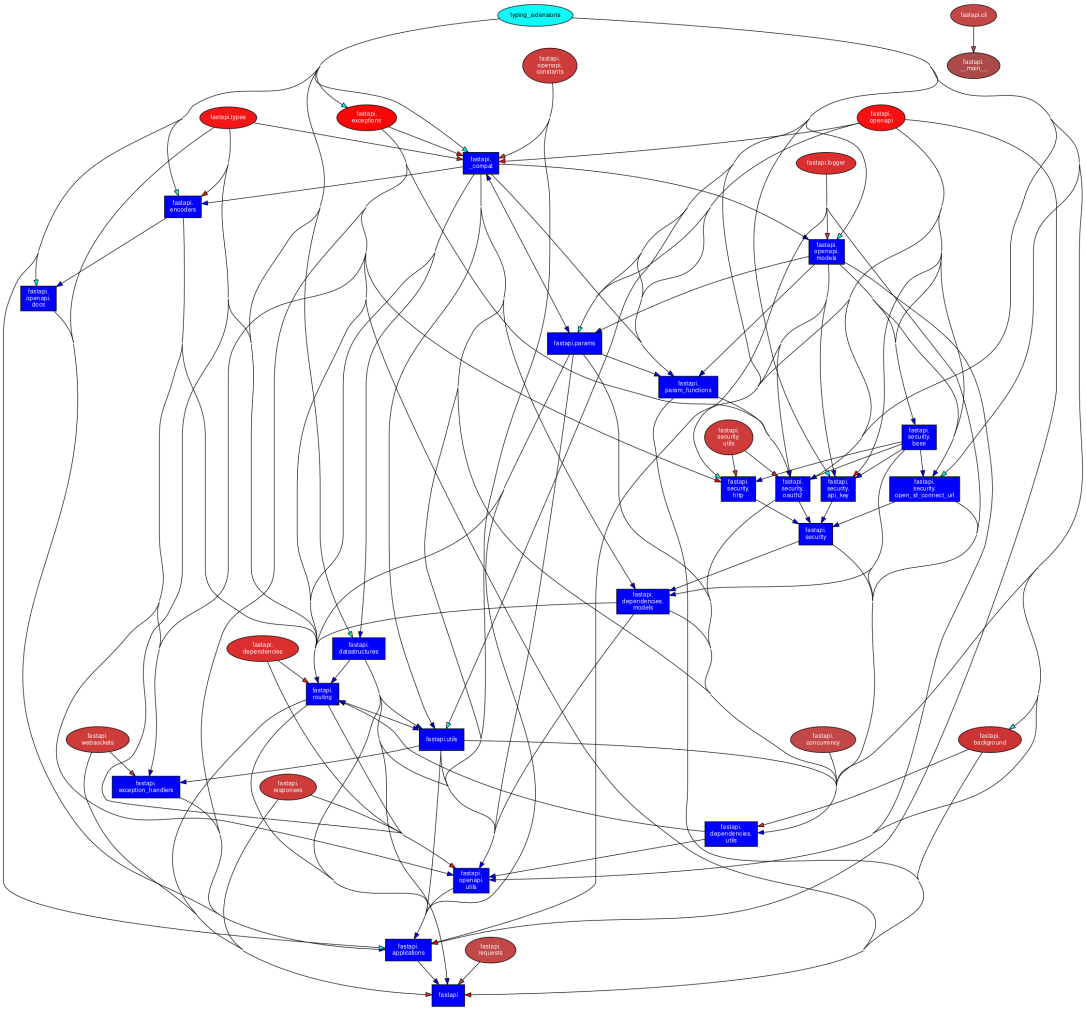


Figure 4: Fastapi Dependency graph

- Fan-in and Fan-out using JSON analysis:

Module	**Fan-in**	**Fan-out**
__main__	0	4
bacon	46	0
fastapi	0	5
fastapi.__main__	0	5
fastapi._compat	0	5
fastapi.applications	0	5
fastapi.background	0	5
fastapi.cli	0	4
fastapi.concurrency	0	4
fastapi.datastructures	0	5
fastapi.dependencies	0	4
fastapi.dependencies.models	0	5
fastapi.dependencies.utils	0	5
fastapi.encoders	0	5
fastapi.exception_handlers	0	5
fastapi.exceptions	0	5
fastapi.logger	0	4
fastapi.middleware	0	4
fastapi.middleware.cors	0	4
fastapi.middleware.gzip	0	4
fastapi.middleware.httpsredirect	0	4
fastapi.middleware.trustedhost	0	4
fastapi.middleware.wsgi	0	4
fastapi.openapi	0	4
fastapi.openapi.constants	0	4
fastapi.openapi.docs	0	5
fastapi.openapi.models	0	5
fastapi.openapi.utils	0	5
fastapi.param_functions	0	5
fastapi.params	0	5
fastapi.requests	0	4
fastapi.responses	0	4
fastapi.routing	0	5
fastapi.security	0	5
fastapi.security.api_key	0	5
fastapi.security.base	0	5
fastapi.security.http	0	5
fastapi.security.oauth2	0	5
fastapi.security.open_id_connect_url	0	5
fastapi.security.utils	0	4
fastapi.staticfiles	0	4
fastapi.templating	0	4
fastapi.testclient	0	4
fastapi.types	0	4
fastapi.utils	0	5
fastapi.websockets	0	4
imported_by	45	0
imports	25	0
name	46	0
path	46	0
typing_extensions	0	4

Table 1: Fan-in and Fan-out Analysis of Modules

- There are no cyclic dependencies in the above graph.

- There are no unused or disconnected modules in the above graph.
- The maximum depth of the dependeny graph is 3.

2.4 Dependency Impact Assessment

- `Bacon` is the core module and changing it would result in 46 dependencies getting affected.
- `Bacon` has 24 risky modules based on the degree of the node.

2.5 Cohesion Analysis Using LCOM (Java)

- Running the LCOM analysis using the command:

```
java {jar LCOM.jar -i jenkins/ -o lcom_analysis/
```

- Analyzed the CSV output for Top 5 Packages in each LCOM1-LCOM5 and YALCOM:

Project Name	Package Name	Type Name	LCOM1
jenkins	jenkins.model	Jenkins	81113.0
jenkins	hudson	FilePath	37529.0
jenkins	hudson.model	Queue	21171.0
jenkins	hudson	Functions	19628.0
jenkins	hudson.model	UpdateCenter	16497.0

Table 2: Top 5 Classes for LCOM1

Project Name	Package Name	Type Name	LCOM2
jenkins	jenkins.model	Jenkins	79605.0
jenkins	hudson	FilePath	31987.0
jenkins	hudson.model	Queue	19764.0
jenkins	hudson	Functions	19555.0
jenkins	hudson.model	UpdateCenter	15039.0

Table 3: Top 5 Classes for LCOM2

Project Name	Package Name	Type Name	LCOM3
jenkins	jenkins.model	Jenkins	219.0
jenkins	hudson	Functions	175.0
jenkins	jenkins.security.stapler	DoActionFilterTest	125.0
jenkins	hudson	FilePath	112.0
jenkins	hudson.model	Queue	105.0

Table 4: Top 5 Classes for LCOM3

Project Name	Package Name	Type Name	LCOM4
jenkins	hudson	Functions	139.0
jenkins	jenkins.security.stapler	DoActionFilterTest	125.0
jenkins	jenkins.model	Jenkins	123.0
jenkins	jenkins.security.stapler	GetterMethodFilterTest	85.0
jenkins	hudson.model	Queue	82.0

Table 5: Top 5 Classes for LCOM4

Project Name	Package Name	Type Name	LCOM5
jenkins	hudson	ThreadGroupMap	2.0
jenkins	hudson.model	MockBuilderThrowsError	2.0
jenkins	hudson.util	FormValidationTest	2.0
jenkins	jenkins	ExtensionFilterTest	2.0
jenkins	hudson.cli	CancelQuietDownCommand	2.0

Table 6: Top 5 Classes for LCOM5

Project Name	Package Name	Type Name	YALCOM
jenkins	hudson.util	FormValidationTest	1.0
jenkins	hudson.model.labels	LabelBenchmark	1.0
jenkins	hudson.model.labels	NodeLabelBenchmark	1.0
jenkins	jenkins.util	JenkinsJVMTest	1.0
jenkins	hudson.model.labels	LabelBenchmarkTest	1.0

Table 7: Top 5 Classes for YALCOM

2.6 Analysis of High LCOM Values in Classes

2.6.1 What does a high LCOM value suggest about a class's design?

A high **Lack of Cohesion of Methods (LCOM)** value generally indicates poor class design and suggests that:

- The methods in the class operate on **disjoint sets of instance variables**, meaning they do not share a common state.
- The class might be **trying to do too many things**, violating the **Single Responsibility Principle (SRP)**.
- There could be **unrelated functionalities** bundled together, making the class harder to maintain, test, and extend.
- A high LCOM value often suggests that the class is a **"God Class"**, which accumulates too much responsibility.

For example, in our analysis:

- `Jenkins` (**LCOM1 = 81113.0**, **LCOM2 = 79605.0**)
- `FilePath` (**LCOM1 = 37529.0**, **LCOM2 = 31987.0**)
- `Queue` (**LCOM1 = 21171.0**, **LCOM2 = 19764.0**)

These classes have extremely high LCOM values, which suggests they are performing multiple unrelated tasks.

2.6.2 Is there a chance for performing functional decomposition?

Yes, functional decomposition can **improve the cohesion** of these classes by breaking them into smaller, more focused classes. Some strategies include:

- **Extracting Classes:** If different methods work on different subsets of instance variables, consider splitting them into separate classes.
- **Identifying Responsibilities:** Each class should follow the **Single Responsibility Principle (SRP)**. If a class handles multiple concerns, separate them into different classes.

- **Refactoring with Design Patterns:**
  - **Facade Pattern:** If the class is managing multiple functionalities, a Facade can simplify interactions.
  - **Strategy Pattern:** If the class has multiple independent behaviors, encapsulating them as separate strategies can help.
  - **Factory Pattern:** If object creation logic is mixed with behavior, a factory can separate concerns.
- **Grouping Related Methods:** If possible, group related methods into inner classes or helper classes.

For instance:

- The `Jenkins` class likely has methods related to **configuration, execution, logging, and security**, which could be **split into multiple cohesive classes**.
- The `FilePath` class might mix **file system operations and security checks**, which should be **separated**.
- The `Queue` class might handle multiple types of jobs, which could be **divided** into separate queue managers.

3 Results and Analysis

3.1 Dependency Analysis (Python)

The dependency analysis using `pydeps` provided a comprehensive view of the module interactions within the selected Python repository (FastAPI). The key findings are as follows:

- The generated dependency graph shows that most modules have a shallow dependency depth (maximum depth = 3), indicating that the codebase is organized in a flat structure.
- The Fan-in and Fan-out analysis revealed:
  - Certain modules (e.g., `bacon`) exhibit high fan-in values, meaning they are widely depended upon across the project.
  - Many modules have a fan-out value of 4 or 5, which suggests that they invoke a consistent set of functions or classes from other modules.
- No cyclic dependencies were detected, and all modules appear to be connected, implying that there are no isolated or unused components.
- The overall coupling in the project is moderate, though the high fan-in for some core modules (like `bacon`) indicates potential risk areas. A change in these core modules may have a cascading effect on a large portion of the system.

3.2 Cohesion Analysis (Java)

The LCOM analysis on the Java project (Jenkins) provided insight into the cohesion levels of different classes:

- The top 5 classes for each LCOM metric (LCOM1–LCOM5 and YALCOM) were extracted from the analysis output. The results indicate that several classes, such as `Jenkins`, `FilePath`, and `Queue`, have extremely high LCOM1 and LCOM2 values.

- A high LCOM value suggests that the methods in a class operate on disjoint sets of instance variables. This indicates that the class is likely handling multiple unrelated responsibilities, thereby violating the Single Responsibility Principle.
- The presence of such high LCOM values highlights the potential for performing functional decomposition. For example:
  - The `Jenkins` class, with LCOM1 = 81113.0 and LCOM2 = 79605.0, likely contains diverse functionalities such as configuration, execution, logging, and security, which could be split into smaller, more cohesive classes.
  - The `FilePath` and `Queue` classes also show high values, suggesting they are doing more than one thing and might benefit from refactoring.
- Refactoring strategies such as extracting helper classes, using design patterns (Facade, Strategy, or Factory), or simply grouping related methods into separate components can help improve cohesion and make the code more maintainable.

3.3 Comparative Table of LCOM Metrics

Below is a table summarizing the LCOM metrics for a set of classes from the Java project:

Package Name	Type Name	LCOM1	LCOM2	LCOM3	LCOM4	LCOM5	YALCOM
jenkins.websocket	Jetty12EE9Provider	150.0	147.0	16.0	15.0	0.9559	0.7778
jenkins.websocket	WriteCallbackImpl	0.0	0.0	1.0	1.0	-0.0	0.0
jenkins.websocket	Provider	66.0	0.0	12.0	12.0	0.0	-1.0
jenkins.websocket	Listener	15.0	0.0	6.0	6.0	0.0	-1.0
jenkins.websocket	Handler	10.0	0.0	5.0	5.0	0.0	-1.0

Table 8: LCOM Metrics for Selected Classes in the Jenkins Websocket Package

4 Discussion and Conclusion

4.1 Challenges and Reflections

- Setting up the analysis tools required careful configuration, particularly for handling large dependency graphs with `pydeps`.
- The complexity of cyclic dependencies and high coupling in certain modules posed challenges in interpreting the results.
- LCOM analysis highlighted several classes with design flaws that may benefit from refactoring.

4.2 Lessons Learned

- Dependency graphs are invaluable for understanding module interrelationships and identifying potential refactoring opportunities.
- High coupling and low cohesion are strong indicators of design flaws that could affect long-term maintainability.
- Utilizing multiple tools (e.g., `pydeps` for Python and LCOM for Java) provides a more holistic view of software quality.

4.3 Summary

This lab provided practical insights into module dependency and cohesion analysis. The combination of `pydeps` for dependency analysis and LCOM metrics for class cohesion enabled a thorough evaluation of both Python and Java projects. The findings not only identified critical design flaws but also suggested targeted refactoring opportunities to improve overall system maintainability.

5 Appendix

5.1 Commands Used

```
# Generate dependency graph using pydeps
pydeps . --show-deps > dependencies.json

# Run LCOM analysis on Java project
java -jar LCOM.jar -i <input_path> -o <output_path>
```

analysis.py

```python
import json
import networkx as nx
from collections import defaultdict

def load_dependencies(json_file):
    with open(json_file, 'r') as f:
        data = json.load(f)
    return data

def build_dependency_graph(dependencies):
    G = nx.DiGraph()
    fan_in = defaultdict(int)
    fan_out = defaultdict(int)

    for module, deps in dependencies.items():
        for dep in deps:
            G.add_edge(dep, module)
            fan_out[module] += 1
            fan_in[dep] += 1

    return G, fan_in, fan_out

def analyze_coupling(fan_in, fan_out):
    all_modules = sorted(set(fan_in.keys()).union(set(fan_out.keys())))
    print("Highly Coupled Modules:")
    print("{:<50} | {:<10} | {:<10}".format("Module", "Fan-in", "Fan-out"))
    print("-" * 75)
    for module in all_modules:
        print("{:<50} | {:<10} | {:<10}".format(module, fan_in.get(module,
        ↪  0), fan_out.get(module, 0)))

def detect_cycles(G):
```

```
    try:
        cycles = list(nx.simple_cycles(G))
        if cycles:
            print("Cyclic Dependencies Detected:")
            for cycle in cycles:
                print(" -> ".join(cycle))
        else:
            print("No cyclic dependencies found.")
    except Exception as e:
        print("Error detecting cycles:", e)

def find_disconnected_modules(G):
    disconnected = [node for node in G.nodes if G.in_degree(node) == 0 and
    ↪  G.out_degree(node) == 0]
    if disconnected:
        print("Unused/Disconnected Modules:", disconnected)
    else:
        print("No unused modules detected.")

def dependency_depth(G):
    if len(G.nodes) == 0:
        print("Graph is empty.")
        return
    depths = {node: nx.single_source_shortest_path_length(G, node) for node
    ↪  in G.nodes}
    max_depth = max(max(depth.values(), default=0) for depth in
    ↪  depths.values())
    print(f"Maximum Dependency Depth: {max_depth}")

def identify_core_module(fan_in):
    if not fan_in:
        return None
    return max(fan_in, key=fan_in.get)

def impact_analysis(G, core_module):
    if core_module not in G:
        print(f"Module {core_module} not found in dependencies.")
        return

    affected_modules = nx.descendants(G, core_module)
    risky_modules = [mod for mod in affected_modules if G.in_degree(mod) > 4]

    print(f"Changes in {core_module} affect {len(affected_modules)}
    ↪  modules:", affected_modules)
    print(f"Risky Modules (high impact if modified) ({len(risky_modules)}):",
    ↪  risky_modules)

if __name__ == "__main__":
    json_path = "dependencies.json"
    dependencies = load_dependencies(json_path)
```

```
    G, fan_in, fan_out = build_dependency_graph(dependencies)

    analyze_coupling(fan_in, fan_out)
    detect_cycles(G)
    find_disconnected_modules(G)
    dependency_depth(G)

    core_module = identify_core_module(fan_in)
    if core_module:
        print(f"Identified Core Module: {core_module}")
        impact_analysis(G, core_module)
    else:
        print("No core module identified.")
```

5.2 Additional Resources

- pydeps GitHub Repository
- LCOM on GitHub

Lab Assignment Report 10

CS202: Software Tools and Techniques for CSE

Roll Number: 22110087
Name: Parth Govale

Contents

1 Introduction, Setup, and Tools **2**
1.1 Overview . . . 2

2 Environment Setup **2**
2.1 System Requirements . . . 2

3 Methodology and Execution **2**
3.1 Setting Up the .NET Development Environment . . . 2
3.2 Understanding Basic Syntax and Control Structures . . . 2
3.3 Implementing Loops and Functions . . . 3
3.4 Object-Oriented Programming in C# . . . 4
3.5 Exception Handling . . . 6
3.6 Debugging Using Visual Studio Debugger . . . 6

4 Results and Analysis **8**
4.1 Activity 2: Basic Arithmetic and Control Structures . . . 8
4.2 Activity 3: Loops and Factorial Function . . . 9
4.3 Activity 4: Object-Oriented Programming . . . 10
4.4 Activity 5: Exception Handling . . . 11

5 Discussion and Conclusion **11**
5.1 Challenges and Reflections . . . 11
5.2 Lessons Learned . . . 12
5.3 Summary . . . 12

6 Appendix **12**
6.1 Tools and Resources . . . 12

1 Introduction, Setup, and Tools

1.1 Overview

This assignment focuses on creating simple console applications to understand basic syntax, control structures, and object-oriented programming principles in C#. In this lab, we will set up the development environment using Visual Studio, write and execute basic C# console applications, implement loops and functions, apply object-oriented programming concepts, handle exceptions, and utilize the Visual Studio Debugger.

2 Environment Setup

2.1 System Requirements

- **Operating System:** Windows 11
- **Software:** Visual Studio 2022 (Community Edition) with .NET SDK
- **Programming Language:** C# (latest stable version)

3 Methodology and Execution

3.1 Setting Up the .NET Development Environment

- Open Visual Studio 2022 and create a new C# Console Application project.
- Ensure that the target framework is set to .NET 6 (or later).
- Write and run a simple "Hello World" program to verify that the environment is properly configured.

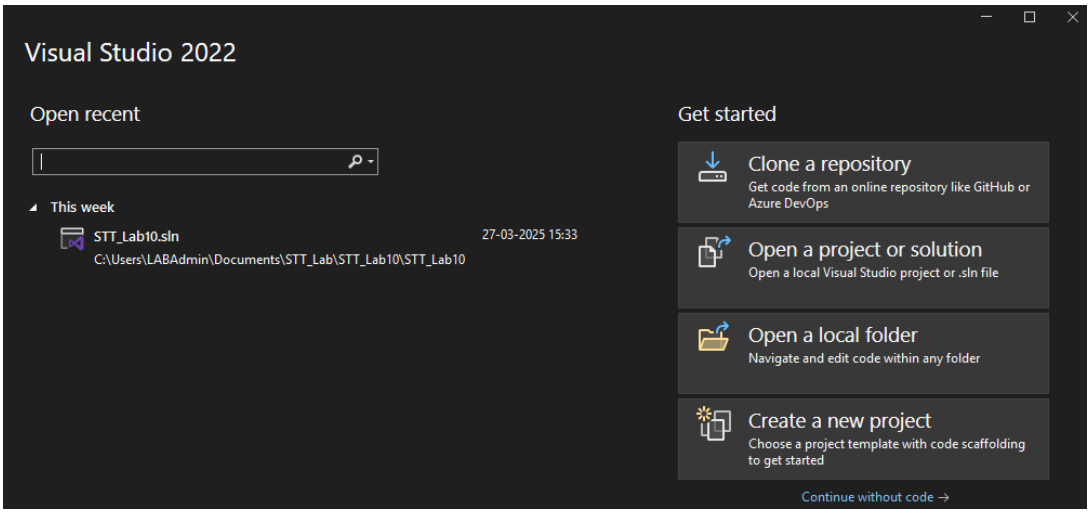


Figure 1: Visual Studio Code 2022

3.2 Understanding Basic Syntax and Control Structures

- Develop an object-oriented C# console application that:
  - Accepts user input for two numbers.

- Performs addition, subtraction, multiplication, and division.
- Uses `if-else` conditions to determine if the sum is even or odd.
- Displays the results using `Console.WriteLine()`.

```
// Activity 2: Basic Arithmetic and Control Structures
using System;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter first number:");
            double num1 = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Enter second number:");
            double num2 = Convert.ToDouble(Console.ReadLine());

            double sum = num1 + num2;
            double diff = num1 - num2;
            double prod = num1 * num2;
            double quot = (num2 != 0) ? num1 / num2 : 0;

            Console.WriteLine($"Sum: {sum}");
            Console.WriteLine($"Difference: {diff}");
            Console.WriteLine($"Product: {prod}");
            if (num2 == 0)
            {
                Console.WriteLine("Division: Cannot divide by zero.");
            }
            else
            {
                Console.WriteLine($"Quotient: {quot}");
            }

            if (sum % 2 == 0)
                Console.WriteLine("The sum is even.");
            else
                Console.WriteLine("The sum is odd.");
        }
    }
}
```

3.3 Implementing Loops and Functions

- Create a C# program that:
  - Uses a `for` loop to print numbers from 1 to 10.
  - Uses a `while` loop to repeatedly prompt the user for input until they type `"exit"`.
  - Defines and calls a function that calculates the factorial of a number provided by the user.

```
// Activity 3: Loops and Factorial Function
using System;

namespace LoopAndFunctionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            // For loop: Print numbers from 1 to 10
            for (int i = 1; i <= 10; i++)
            {
                Console.WriteLine(i);
            }

            // While loop: Continue until user types "exit"
            string input = "";
            while (!input.Equals("exit", StringComparison.OrdinalIgnoreCase))
            {
                Console.WriteLine("Enter a number (or type 'exit' to quit):");
                input = Console.ReadLine();
            }

            // Factorial Function
            Console.WriteLine("Enter a number to compute its factorial:");
            int n = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine($"Factorial of {n} is {Factorial(n)}");
        }

        static long Factorial(int n)
        {
            if (n <= 1)
                return 1;
            return n * Factorial(n - 1);
        }
    }
}
```

3.4 Object-Oriented Programming in C#

- Create a class `Student` with:
  - Properties: `Name`, `ID`, `Marks` (use appropriate C# data types).
  - A constructor to initialize these values.
  - A copy-constructor and/or overloaded constructors as needed.
  - A method `GetGrade()` that returns a grade based on the marks (e.g., A, B, C, etc.).
  - A static `StudentIITGN()` method to create and display student details.
- Create a subclass `StudentIITGN` derived from `Student` with:
  - An additional property: `Hostel_Name_IITGN`.
  - Its own static `Main()` method to create and display details of IITGN students.

```
// Activity 4: Object-Oriented Programming

using System;

namespace StudentApp
{
    class Student
    {
        public string Name { get; set; }
        public int ID { get; set; }
        public double Marks { get; set; }

        // Constructor to initialize student details.
        public Student(string name, int id, double marks)
        {
            Name = name;
            ID = id;
            Marks = marks;
        }

        // Copy Constructor
        public Student(Student other)
        {
            Name = other.Name;
            ID = other.ID;
            Marks = other.Marks;
        }

        // Method to determine grade based on Marks.
        public string GetGrade()
        {
            if (Marks >= 90)
                return "A";
            else if (Marks >= 80)
                return "B";
            else if (Marks >= 70)
                return "C";
            else if (Marks >= 60)
                return "D";
            else
                return "F";
        }
    }

    class StudentIITGN : Student
    {
        public string Hostel { get; set; }

        // Constructor for StudentIITGN including new property.
        public StudentIITGN(string name, int id, double marks, string hostel)
            : base(name, id, marks)
        {
            Hostel = hostel;
        }
    }

    class Program
```

```
    {
        static void Main(string[] args)
        {
            Student s = new Student("Alice", 101, 85);
            Console.WriteLine($"Student: {s.Name}, Grade: {s.GetGrade()}");

            StudentIITGN sIITGN = new StudentIITGN("Bob", 102, 78, "Hostel-1");
            Console.WriteLine($"Student: {sIITGN.Name}, Grade: {sIITGN.GetGrade()},
            ↪  Hostel: {sIITGN.Hostel}");
        }
    }
}
```

3.5 Exception Handling

- Modify the arithmetic program (from the basic syntax activity) to handle exceptions:
  - Use a `try-catch` block to handle division-by-zero errors.
  - Ensure that the program does not crash on invalid user input.

```
// Activity 5: Exception Handling
using System;

namespace ExceptionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Enter dividend:");
                double dividend = Convert.ToDouble(Console.ReadLine());
                Console.WriteLine("Enter divisor:");
                double divisor = Convert.ToDouble(Console.ReadLine());
                double result = dividend / divisor;
                Console.WriteLine($"Result: {result}");
            }
            catch (DivideByZeroException)
            {
                Console.WriteLine("Error: Division by zero is not allowed.");
            }
            catch (Exception ex)
            {
                Console.WriteLine($"An error occurred: {ex.Message}");
            }
        }
    }
}
```

3.6 Debugging Using Visual Studio Debugger

After inserting breakpoints in your code, follow these steps to perform debugging:

1. **Start Debugging (F5):** Press `F5` or click the "Start Debugging" button in Visual Studio. The application will compile and run in Debug mode, and execution will pause at the first breakpoint encountered.

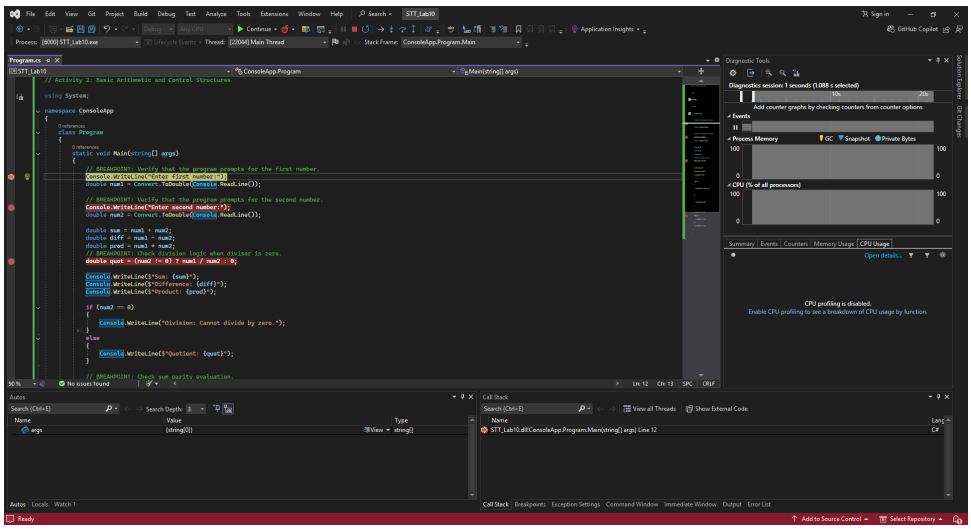


Figure 2: Visual Studio Debugger - Start Debugging

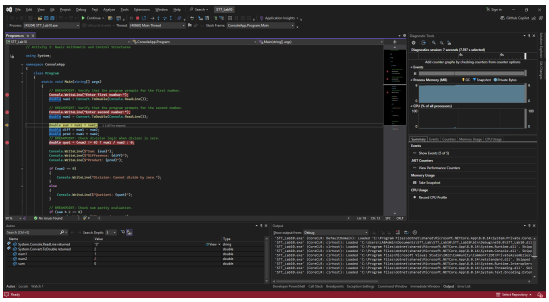
2. **Step Over (F10):** Use `F10` to execute the current line without entering any called functions. This is useful when you want to move quickly to the next line while ignoring the internal details of a function.

3. **Step In (F11):** Press `F11` to step into a function or method call. This lets you examine the execution flow inside the called function, so you can inspect its internal logic and variable values.

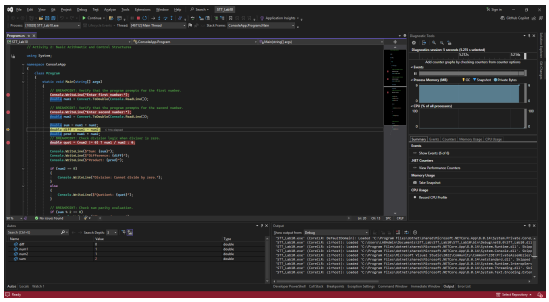
4. **Step Out (Shift+F11):** If you are inside a function and wish to return to the calling function, press `Shift+F11` to step out. This command completes the execution of the current function and returns control to the code that called it.

4 Results and Analysis

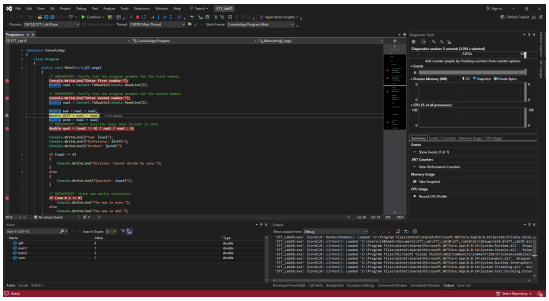
4.1 Activity 2: Basic Arithmetic and Control Structures



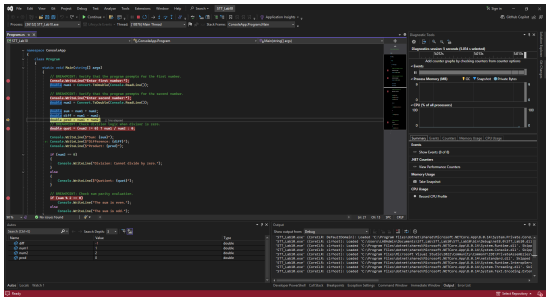
Step Over (Previous)



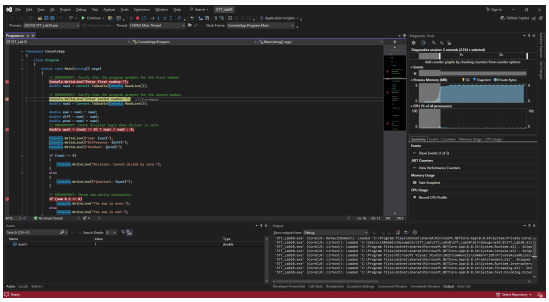
Step Over (Next)



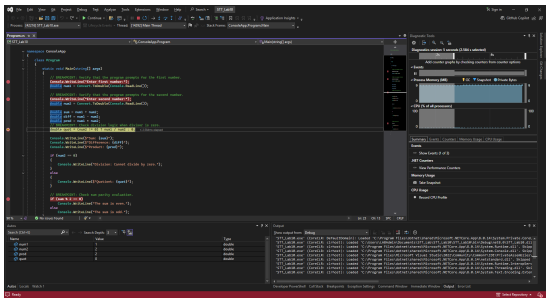
Step In (Previous)



Step In (Next)



Step Out (Previous)

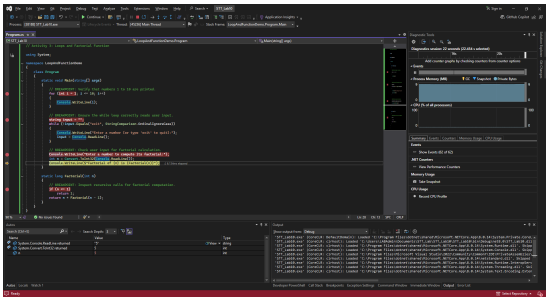


Step Out (Next)

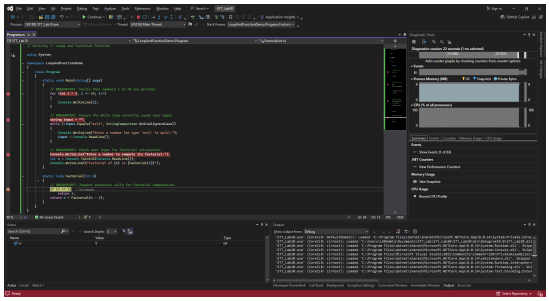
Figure 3: Debugging Visual Illustrations for Activity 2: Basic Arithmetic and Control Structures

- **Step Over:** The images show the code executing one line at a time without entering function calls.
- **Step In:** The images reveal the debugger executing similar to step over as there is no function call.
- **Step Out:** The images show the debugger jumping from one breakpoint to another.

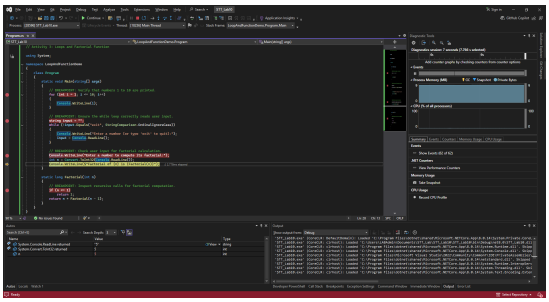
4.2 Activity 3: Loops and Factorial Function



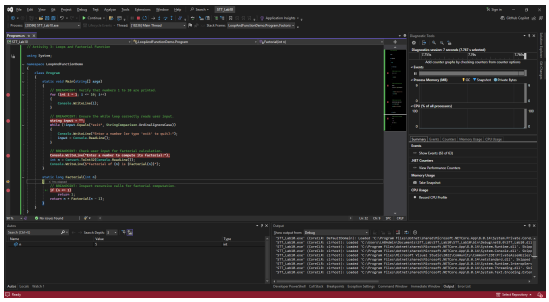
Step Over (Previous)



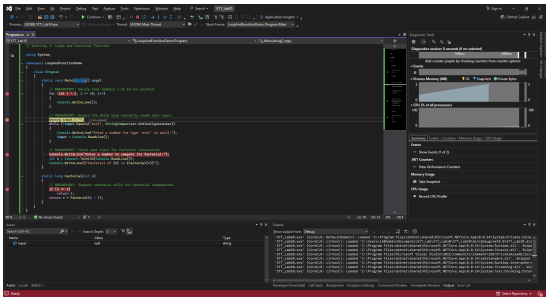
Step Over (Next)



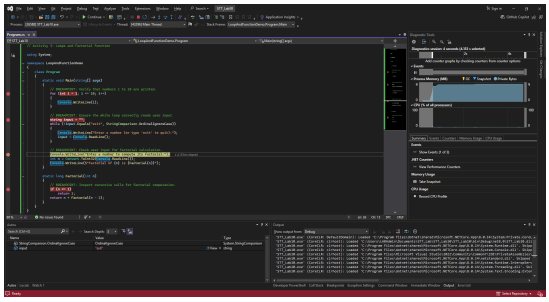
Step In (Previous)



Step In (Next)



Step Out (Previous)

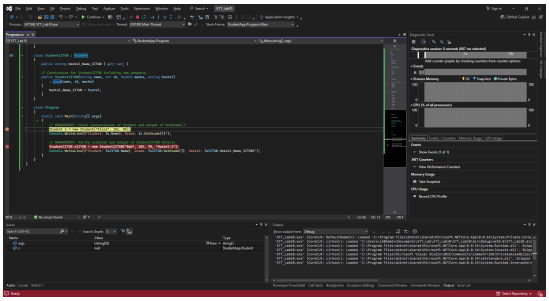


Step Out (Next)

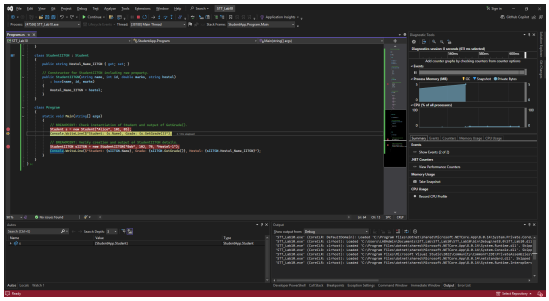
Figure 4: Debugging Visual Illustrations for Activity 3: Loops and Factorial Function

- **Step Over:** The images capture the flow from function call to the breakpoint within the function, skipping a few lines.
- **Step In:** The images display the debugger stepping into the recursive calls of the Factorial function line by line.
- **Step Out:** The images show the debugger exiting the recursive function after completing the calculations, only jumping to breakpoints.

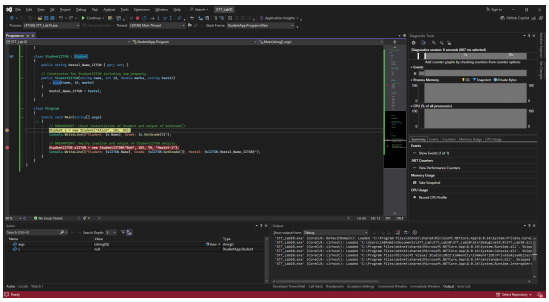
4.3 Activity 4: Object-Oriented Programming



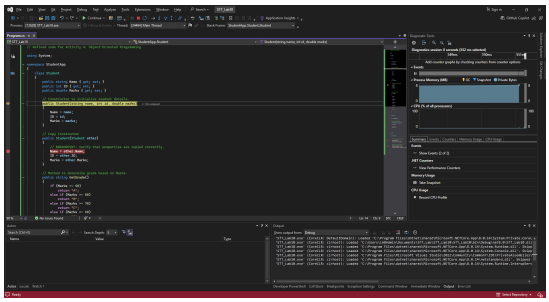
Step Over (Previous)



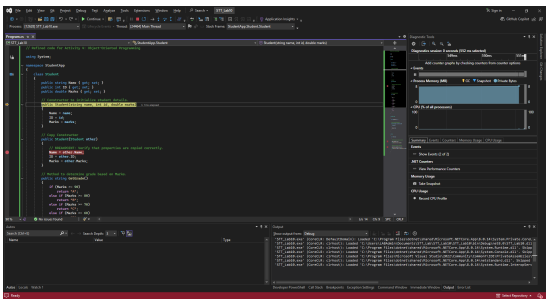
Step Over (Next)



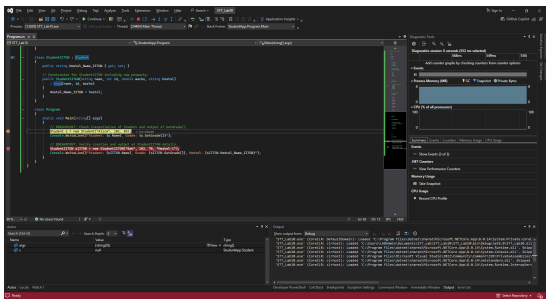
Step In (Previous)



Step In (Next)



Step Out (Previous)

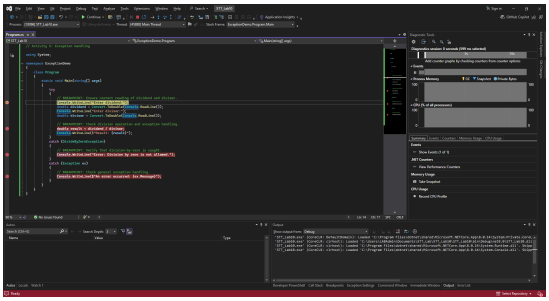


Step Out (Next)

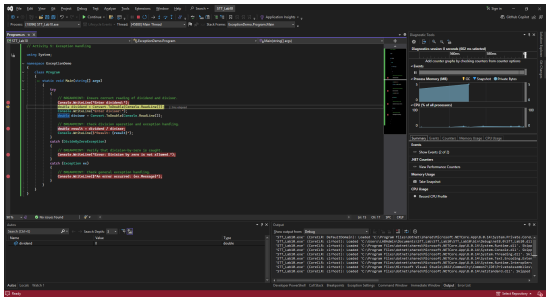
Figure 5: Debugging Visual Illustrations for Activity 4: Object-Oriented Programming

- **Step Over:** The images demonstrate the creation of `Student` and `StudentIITGN` objects without inspecting inside constructors.
- **Step In:** The images capture the debugger entering the constructors method to check property values line by line.
- **Step Out:** The images show the exit from these functions back to the main method and to the next breakpoint.

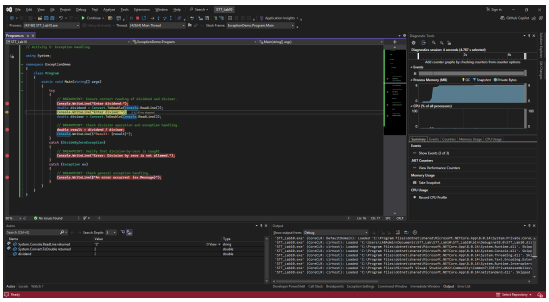
4.4 Activity 5: Exception Handling



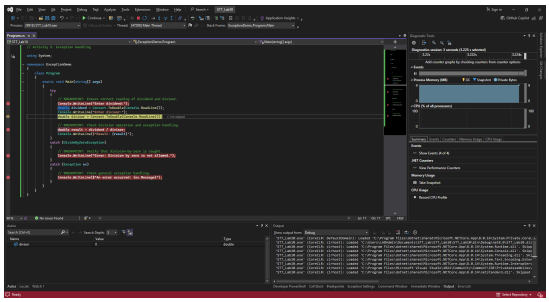
Step Over (Previous)



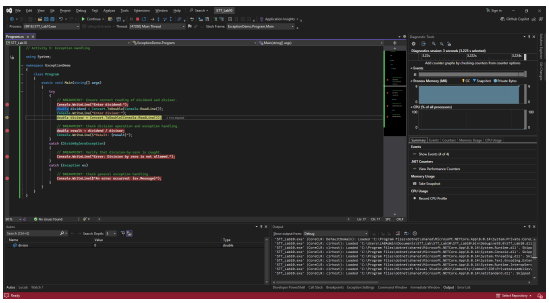
Step Over (Next)



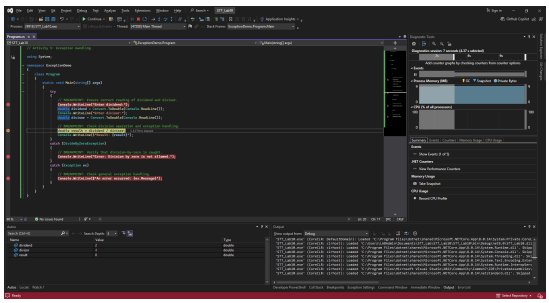
Step In (Previous)



Step In (Next)



Step Out (Previous)



Step Out (Next)

Figure 6: Debugging Visual Illustrations for Activity 5: Exception Handling

- **Step Over:** The images show the code executing the try block, handling user input and arithmetic operations.
- **Step In:** The images show similar output to step over as no function calling is happening.
- **Step Out:** The images capture the debugger returning to the main flow and jumping to the next breakpoint.

5 Discussion and Conclusion

5.1 Challenges and Reflections

- Configuring Visual Studio and ensuring compatibility with the .NET SDK required careful attention.
- Incorporating exception handling to manage various input errors was challenging, but necessary for building robust applications.

5.2 Lessons Learned

- A well-configured development environment is essential for efficient coding and debugging.
- Exception handling and debugging are critical for developing stable software.
- Object-oriented programming concepts, such as encapsulation and inheritance, significantly enhance code modularity and maintainability.

5.3 Summary

This lab provided a comprehensive introduction to .NET development using C#. Through the creation of multiple console applications, students gained practical experience with basic syntax, control structures, loops, functions, object-oriented programming, exception handling, and debugging.

6 Appendix

6.1 Tools and Resources

Provide links to any resources or tools that were used for this lab, such as:

- C# Documentation
- Visual Studio Official Website
- .NET SDK Downloads
- Exception Handling in C#
- Visual Studio Debugger Documentation
- C# Inheritance Tutorial