5-Day Software Architecture Curriculum

Audience: Junior Engineers **Format:** 5 Days, 6 Hours/Day

Methodology: First-Principles Reconstruction. Instead of relying on pre-existing static analysis tools that treat metrics as opaque "black boxes," you will build the verification tools yourself. By writing the code to calculate complex metrics—such as parsing Abstract Syntax Trees (AST) to measure Coupling or using Reflection to map Inheritance depth—you will gain a fundamental understanding of *why* code quality fails. You will reconstruct the mathematical models (Graph Theory for complexity, Set Theory for cohesion) from scratch to see the true physics of software architecture.

Abbreviations & Glossary

Abbreviation	Full Term	Explanation
A	Abstractness	A metric measuring the ratio of abstract classes/interfaces to total classes in a package ($A = N_{abst}/N_{total}$).
AST	Abstract Syntax Tree	A tree representation of the abstract syntactic structure of source code, often used by compilers and static analysis tools.
Ca	Afferent Coupling	The number of classes outside a package that depend on classes inside the package ("Incoming" dependencies). High Ca implies responsibility.
CBO	Coupling Between Object classes	The count of the number of other classes to which a class is coupled via method calls, field access, inheritance, etc.
CC	Cyclomatic Complexity	A quantitative measure of the number of linearly independent paths through a program's source code.
Ce	Efferent Coupling	The number of classes inside a package that depend on classes outside the package ("Outgoing" dependencies). High Ce implies vulnerability.
CFG	Control Flow Graph	A representation, using graph notation, of all paths that might be traversed through a program during its execution.
CK	Chidamber & Kemerer	A suite of six metrics proposed in 1994 to measure the design quality of object-oriented software.
CoN	Connascence	A modern metric measuring the strength of coupling. Two components are connascent if a change in one requires a change in the other.
D	Distance from Main Sequence	A metric indicating balance between Abstractness and Instability: $D = \|A + I - 1$.
DIT	Depth of Inheritance Tree	The length of the longest path from a class to the root of the inheritance hierarchy.
I	Instability	A ratio calculated as $I = Ce/(Ca + Ce)$. Indicates the resilience of a package to change (0 = Stable, 1 = Unstable).
LCOM	Lack of Cohesion of Methods	A measure of how disparate the methods in a class are, calculated by analyzing shared field usage among methods.
LOC	Lines of Code	A software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code.
MI	Maintainability Index	A composite metric calculated from Halstead Volume, Cyclomatic Complexity, and LOC.
NOC	Number of Children	The count of immediate subclasses subordinated to a class in the class hierarchy.
OOD	Object-Oriented Design	The process of planning a system of interacting objects for the purpose of solving a software problem.
OOP	Object-Oriented Programming	A programming paradigm based on the concept of "objects", which can contain data and code.
RFC	Response For a Class	The set of methods that can potentially be executed in response to a message received by an object of that class (Internal methods + External calls).
V	Halstead Volume	A metric reflecting the information content (bits) of the code based on the number of operators and operands.
WMC	Weighted Methods per Class	The sum of the complexity of all methods in a class.

Day 1 Explore The Physics of Objects including Polymorphism & Cohesion

We dedicate the first day to understanding the fundamental memory layouts, Dynamic Dispatch mechanisms, and the LCOM metrics that define object cohesion.

Hours 1-2 Examine The Cost of Abstraction and OOP Internals

We begin by studying the Virtual Method Table (vtable) and Object Header layout in C# to gain a deep understanding of the memory overhead associated with a class versus a struct.

It is crucial to address the **Microbenchmarking Trap**. A common fallacy among developers is that naive measurements of `virtual` versus `non-virtual` calls accurately reflect performance costs. In reality, modern JIT compilers (such as Dotnet 6+ and JVM) utilize advanced techniques like **Devirtualization**, **Tiered Compilation**, and **PGO (Profile-Guided Optimization)**. If the compiler detects that you are strictly using one implementation in a loop, it will often "devirtualize" the call into a direct pointer jump or inline it entirely. Therefore, the lesson is that while abstraction isn't free, it is *sometimes* optimized away completely.

For your coding assignment, you will write a program using a professional harness like **BenchmarkDotNet** (C#) or **JMH** (Java). Your challenge is to construct a scenario that explicitly *defeats* the optimizer. You can achieve this by using Reflection or a random number generator to instantiate different subclasses at runtime, forcing the CPU to perform a true v-table lookup. You will then compare this against the "optimized" static path.

Hours 3-4 Cover The Chidamber & Kemerer (CK) Metric Suite

We spend this session mastering the six foundational metrics defined by Chidamber and Kemerer in 1994. These metrics serve as the physics of OOP, allowing us to measure volume, density, and connectivity.

1. Weighted Methods per Class (WMC)

WMC measures complexity by summing the complexity of a class's methods. Originally undefined, we typically calculate this with a weight of 1 per method or by using Cyclomatic Complexity. A high WMC suggests a class is doing too much, which hurts maintainability.

Consider this example of complexity weighting:

C# Example

```
// POOR: High WMC (God Method)
class Processor {
    public void Process() {
        // WMC Analysis:
        // Every 'if' statement adds +1 to Cyclomatic Complexity (CC).
        // A single large method accumulates all CC in one place.
        if(true /*user.isValid*/) {
            if(true /*cart.isNotEmpty*/) {
                // ... 50 lines of payment logic ...
            }
        }
    }
}

// GOOD: Low WMC (Decomposed)
class CleanProcessor {
    public void Process() {
        // WMC Analysis:
        // This method has CC = 1 (Linear flow).
        // It delegates complexity to sub-methods.
        if (!ValidateUser()) return;
        ProcessPayment();
    }

    private bool ValidateUser() { return true; } // CC = 1
    private void ProcessPayment() { /*...*/ }    // CC = 1
    // Total Class WMC is still 3, but average complexity is 1.
}

class Program {
    static void Main() {
        new Processor().Process();
        new CleanProcessor().Process();
    }
}
```

TypeScript Example

```
// POOR: High WMC
class Processor {
    process(): void {
        if (true /* this.user.isValid */) {
            // ... nested logic ...
        }
```

```
    }
}

// GOOD: Low WMC
class CleanProcessor {
    process(): void {
        if (!this.validateUser()) return;
        this.processPayment();
    }

    private validateUser(): boolean { return true; }
    private processPayment(): void { /*...*/ }
}

// Usage
new Processor().process();
new CleanProcessor().process();
```

2. Depth of Inheritance Tree (DIT)

The DIT metric tracks the longest path from a class to the root of the hierarchy. A high DIT increases the risk of the "Fragile Base Class" problem, where changes in a grandparent class break a grandchild class.

Java/C# Example

```
// POOR: High DIT (Deep Inheritance)
class Widget {}
class Button extends Widget {}
class ActionButton extends Button {}
// DIT = 4. Fragile Base Class risk is high.
class SaveButton extends ActionButton {}

// GOOD: Low DIT (Composition)
interface Clickable { void click(); }

// DIT = 1. We implement a contract rather than inheriting state.
class ComposeButton implements Clickable {
    public void click() { /* impl */ }
}

class Program {
    public static void main(String[] args) {
        new SaveButton();
        new ComposeButton().click();
    }
}
```

TypeScript Example

```
// POOR: High DIT
class Widget {}
class Button extends Widget {}
class ActionButton extends Button {}
class SaveButton extends ActionButton {} // DIT = 4

// GOOD: Low DIT
interface IClickable {
    click(): void;
}
class ComposeButton implements IClickable {
    click(): void { console.log("Click"); }
}

// Usage
new SaveButton();
new ComposeButton().click();
```

3. Number of Children (NOC)

NOC counts immediate subclasses. High NOC on a concrete class is risky because changes ripple down to many children. However, high NOC on an interface is excellent, as it indicates a healthy, polymorphic system.

C# Example

```
// POOR: Rigid Base Class with High NOC
class BaseShape {
   // Logic here is dangerous because 2 children depend on it.
   // Changing this logic risks breaking Circle and Square.
```

```
   public void Draw() { Console.WriteLine("Setting generic color"); }
}
class Circle : BaseShape {}
class Square : BaseShape {}

// GOOD: Stable Abstraction with High NOC
// High NOC on an interface is good (Polymorphism).
// The interface defines a contract, not implementation logic.
interface IShape {
    void Draw();
}
class GoodCircle : IShape { public void Draw() { /*...*/ } }
class GoodSquare : IShape { public void Draw() { /*...*/ } }

class Program {
    static void Main() {
        // Usage
        var shapes = new List<BaseShape> { new Circle(), new Square() };
        shapes.ForEach(s => s.Draw());

        var goodShapes = new List<IShape> { new GoodCircle(), new GoodSquare() };
        goodShapes.ForEach(s => s.Draw());
    }
}
```

TypeScript Example

```
// POOR: Rigid Base Class with High NOC
class BaseShape {
   // Logic here is dangerous because 2 children depend on it.
   draw() { console.log("Setting generic color"); }
}
class Circle extends BaseShape {}
class Square extends BaseShape {}

// GOOD: Stable Abstraction with High NOC
// High NOC on an interface is good (Polymorphism).
interface IShape {
    draw(): void;
}
class GoodCircle implements IShape { draw() { /*...*/ } }
class GoodSquare implements IShape { draw() { /*...*/ } }

// Usage
const shapes: BaseShape[] = [new Circle(), new Square()];
shapes.forEach(s => s.draw());

const goodShapes: IShape[] = [new GoodCircle(), new GoodSquare()];
goodShapes.forEach(s => s.draw());
```

4. Coupling Between Object classes (CBO)

CBO counts the number of other classes a class interacts with. High CBO means tight coupling, which prevents reuse and makes testing difficult.

C# Example

```
// POOR: High CBO (Tight Coupling)
class OrderService {
    public void Checkout() {
        // CBO +1: Direct dependency on 'Logger' class.
        var logger = new Logger();
        var db = new SqlDatabase();
        db.Save();
    }
}

// GOOD: Low CBO (Loose Coupling via DI)
class CleanOrderService {
    private readonly ILogger _logger;
    private readonly IDatabase _db;

    public CleanOrderService(ILogger logger, IDatabase db) {
        _logger = logger;
        _db = db;
    }
}

class Program {
    static void Main() {
        new OrderService().Checkout();
        new CleanOrderService(null, null);
```

```
    }
}
```

TypeScript Example

```
// POOR: High CBO
class OrderService {
    checkout(): void {
        const logger = new Logger(); // Hard coupling
        const db = new SqlDatabase();
        db.save();
    }
}

// GOOD: Low CBO
class CleanOrderService {
    constructor(private logger: ILogger, private db: IDatabase) {}

    checkout(): void {
        this.db.save();
    }
}

// Usage
new OrderService().checkout();
new CleanOrderService({ log: ()=>{} }, { save: ()=>{} }).checkout();
```

5. Response For a Class (RFC)

RFC measures the "immediate reach" of a class—methods it owns plus methods it calls. A high RFC indicates a "chatty" object that knows too much about the system's implementation details.

C# Example

```
// POOR: High RFC (Chatty Object)
class OrderProcessor {
    // Internal Method (M=1)
    public void Process() {
        // External Calls (R=5) -> RFC = 6
        _auth.Check();
        _db.Connect();
        _db.Save();
        _email.Send();
        _logger.Log();
    }
}

// GOOD: Low RFC (Facade Pattern)
class CleanProcessor {
    // Internal Method (M=1)
    public void Process() {
        // External Call (R=1) -> RFC = 2
        _facade.HandleTransaction();
    }
}

class Program {
    static void Main() {
        new OrderProcessor().Process();
        new CleanProcessor().Process();
    }
}
```

TypeScript Example

```
// POOR: High RFC
class OrderProcessor {
    process() {
        auth.check();
        db.connect();
        db.save();
        email.send();
        logger.log();
    }
}
// RFC = 6

// GOOD: Low RFC
class CleanProcessor {
```

```
        process() {
            facade.handleTransaction();
        }
    }
    // RFC = 2

    // Usage
    new OrderProcessor().process();
    new CleanProcessor().process();
```

6. Lack of Cohesion of Methods (LCOM)

LCOM4 measures how disparate the methods in a class are. If you graph methods as nodes and shared fields as edges, LCOM4 counts the connected components. A score greater than 1 implies the class is trying to be two different things and should be split.

Java/C# Example

```
    // POOR: High LCOM
    // Graph: [printName] --x-- [connect]
    // LCOM4 = 2 (Two disjoint components).
    class UserManager {
        String name;
        String dbConfig;
        void printName() { System.out.println(name); }
        void connect() { System.out.println(dbConfig); }
    }

    // GOOD: Low LCOM (Split)
    class User {
        String name;
        // LCOM4 = 1
        void printName() { System.out.println(name); }
    }

    class DbConnector {
        String dbConfig;
        // LCOM4 = 1
        void connect() { System.out.println(dbConfig); }
    }

    class Program {
        public static void main(String[] args) {
            // Poor usage: One object doing two things
            UserManager um = new UserManager();
            um.printName();
            um.connect();

            // Good usage: Separation of Concerns
            User u = new User();
            u.printName();

            DbConnector db = new DbConnector();
            db.connect();
        }
    }
```

TypeScript Example

```
    // POOR: High LCOM
    class UserManager {
        name: string = "User";
        dbConfig: string = "DB";

        // Component A
        printName() { console.log(this.name); }
        // Component B (Disjoint)
        connect() { console.log(this.dbConfig); }
    }

    // GOOD: Low LCOM
    class User {
        name: string = "User";
        printName() { console.log(this.name); }
    }

    class DbConnector {
        dbConfig: string = "DB";
        connect() { console.log(this.dbConfig); }
    }

    // Usage
    const um = new UserManager();
```

```
um.printName();
um.connect();

const u = new User();
u.printName();

const db = new DbConnector();
db.connect();
```

Hours 5-6 Involve Deep Review & Refactoring

For this activity, you will construct a "CK-Analyzer" using **Reflection** (for C#) or **ts-morph** (for TypeScript). The goal is to write a tool that automatically calculates these metrics for a sample codebase, giving you a tangible way to audit code quality.

Day 2 Explore the Topology of Code including Coupling & Inheritance

This day focuses on Afferent/Efferent Coupling, Instability, and the fragility of inheritance.

Hours 1-2 Analyze Coupling Metrics & Instability

We define Instability with the formula $I = Ce/(Ca + Ce)$. This helps us visualize which parts of our system are volatile and which are stable.

C# Example

```
// ANALYZING INSTABILITY (I)
// Formula: I = Ce / (Ca + Ce)

// CASE: High Efferent Coupling (Ce)
// This class is "Vulnerable" (I=1.0) because it creates 3 external types.
class Coordinator {
    void DoWork() {
        var a = new AuthSystem(); // Ce += 1
        var b = new Database();   // Ce += 1
        var c = new Emailer();    // Ce += 1
    }
}

class Program {
    static void Main() {
        new Coordinator().DoWork();
    }
}
```

TypeScript Example (The "Import" Trap)

```
// FALLACY WARNING:
// In TypeScript/JS, "coupling" is not just 'new Class()'.
// It is defined by 'import' statements.
// Even if you never instantiate the class, importing a type creates a dependency.

import { AuthSystem } from './Auth'; // Ce += 1
import { Database } from './DB';     // Ce += 1
import { Emailer } from './Email';   // Ce += 1

// Instability (I) = 3 / (0 + 3) = 1.0 (Maximum Instability)
class Coordinator {
    doWork(): void {
        // ...
    }
}

// Usage
new Coordinator().doWork();
```

Your coding assignment is to build a "Dependency Grapher." For TypeScript, you must parse **AST (Abstract Syntax Trees)** to count imports, as simple regex or class checks are insufficient. You should use the **TypeScript Compiler API** or libraries like `madge`.

Hour 2.5 Introduce Connascence (Granular Coupling)

Modern software architecture critiques CBO as a blunt instrument because it only tells you *that* you are coupled, not *how*. To address this, we introduce **Connascence (CoN)**, which measures the strength of coupling.

- **CoN of Name:** Coupled only by method name (Weak/Good).
- **CoN of Type:** Coupled by specific parameter types (Medium).

- **CoN of Meaning/Algorithm:** "If I return -1, the caller must know that means 'User Not Found'." This is invisible, strong coupling (Bad).

Hours 3-4 Examine the Depth of Inheritance Tree (DIT) & The Fragile Base Class

We explore the dangers of inheritance through the "Fragile Base Class" problem.

C# Example

```
class ListManager {
    // FRAGILITY SOURCE:
    // This implementation calls the virtual Add() method loop.
    // If we optimize this to 'AddRangeInternal', derived classes break.
    public virtual void AddRange(IEnumerable<string> items) {
        foreach (var item in items) Add(item);
    }
    public virtual void Add(string item) { /*...*/ }
}

class CountedList : ListManager {
    public int Count = 0;
    // We rely on the Base implementation calling this exactly once per item.
    public override void Add(string item) {
        Count++;
        base.Add(item);
    }
}

class Program {
    static void Main() {
        var list = new CountedList();
        list.AddRange(new[] { "a", "b" });
        Console.WriteLine(list.Count);
    }
}
```

TypeScript Example

```
class ListManager {
    private items: string[] = [];
    add(item: string) { this.items.push(item); }

    // Fragile Method: Relies on internal call to 'add'
    addRange(items: string[]) { items.forEach(i => this.add(i)); }
}

class CountedList extends ListManager {
    count = 0;
    override add(item: string) {
        this.count++;
        super.add(item);
    }
}

// Usage
const list = new CountedList();
list.addRange(["a", "b"]);
console.log(list.count);
```

Hours 5-6 The Main Sequence & Composition over Inheritance

We introduce the concepts of **Abstractness (A)** and the **Main Sequence** graph. The key metric here is the Distance from the Main Sequence: $D = |A + I - 1|$. Ideally, stable components ($I \to 0$) should be abstract ($A \to 1$). We also refactor our fragile code using Composition.

C# Example (Composition Fix)

```
// COMPOSITION FIX (The Decorator Pattern)
class CountedListWrapper : IList<string> {
    private readonly List<string> _inner = new List<string>();

    // ROBUSTNESS: We explicitly implement the Add logic.
    // We do NOT rely on a parent class calling us back.
    public void Add(string item) {
        _inner.Add(item);
    }
    public int Count => _inner.Count;
    // ... other members
}
```

```
class Program {
    static void Main() {
        new CountedListWrapper().Add("test");
    }
}
```

TypeScript Example

```
interface IList<T> { add(item: T): void; }

// COMPOSITION FIX
class CountedListWrapper implements IList<string> {
    private inner: string[] = [];

    // ROBUSTNESS: No hidden inheritance coupling.
    add(item: string): void {
        this.inner.push(item);
    }
}

// Usage
new CountedListWrapper().add("test");
```

Day 3 Focus on Control Flow Analysis and Cyclomatic Complexity

Hours 1-2 Cover Graph Theory & McCabe’s Formula

C# Example

```
class Logic {
    // POOR: High Cognitive Complexity (Nested)
    // Cyclomatic Complexity (CC) = 4
    // Cognitive Complexity is high due to nesting depth.
    public void CheckNested(int x) {
        if (x > 0) {
            if (x < 100) {
                if (x % 2 == 0) {
                    Console.WriteLine("Even Positive Small");
                }
            }
        }
    }

    // GOOD: Low Cognitive Complexity (Flat)
    // CC is STILL 4 (Same # of paths), but easy to read.
    public void CheckFlat(int x) {
        if (x <= 0) return;
        if (x >= 100) return;
        if (x % 2 != 0) return;

        Console.WriteLine("Even Positive Small");
    }
}
```

```
class Program {
    static void Main() {
        var logic = new Logic();
        logic.CheckNested(10);
        logic.CheckFlat(10);
    }
}
```

TypeScript Example

```
class Logic {
    // POOR: High Cognitive Complexity (Nested)
    checkNested(x: number): void {
        if (x > 0) {
            if (x < 100) {
                if (x % 2 === 0) {
                    console.log("Even Positive Small");
                }
            }
        }
    }

    // GOOD: Low Cognitive Complexity (Flat)
    checkFlat(x: number): void {
        if (x <= 0) return;
```

```
        if (x >= 100) return;
        if (x % 2 !== 0) return;

        console.log("Even Positive Small");
    }
}

// Usage
const logic = new Logic();
logic.checkNested(10);
logic.checkFlat(10);
```

Hours 3-4 Explain Basis Path Testing & Path Sensitization

A modern correction to standard CC calculation involves **Path Sensitization**. Standard CC assumes all paths are reachable, but an "Impossible Path" (e.g., `if (x > 5) { if (x < 3) { ... } }`) should ideally be marked as Dead Logic rather than a valid complexity path.

For your assignment, when calculating paths manually, identify logically impossible branches and annotate them.

Hours 5-6 Compare Cognitive Complexity vs. Cyclomatic Complexity

We finish the day by researching **Cognitive Complexity** (as defined by SonarSource). While Cyclomatic Complexity counts branches, Cognitive Complexity adds penalties for *nesting*, providing a much better indication of how understandable the code truly is.

Day 4 Explore The Type System including Covariance & Contravariance

Hours 1-2 Distinguish Invariance, Covariance, and Contravariance

C# Example (The Array Trap)

```
using System;

class Program {
    static void Main() {
        // FALLACY: Array Covariance is "unsafe" at runtime.
        string[] strings = new string[] { "a", "b" };

        // COVARIANCE: string[] IS-A object[].
        // This assignment is allowed in C#, but it opens a hole in type safety.
        object[] objects = strings;

        try {
            // CRASH: ArrayTypeMismatchException
            // We try to put an int into what the runtime knows is strictly a string array.
            objects[0] = 123;
        } catch (Exception ex) {
            Console.WriteLine($"Runtime Safety Check Failed: {ex.GetType().Name}");
        }
    }
}
```

TypeScript Example (Structural Typing Nuance)

```
class Animal { name: string = ""; }
class Dog extends Animal { bark() { console.log("Woof"); } }

function main() {
    // TypeScript arrays are covariant.
    const dogs: Dog[] = [new Dog()];

    // Assignment allowed because Array<Dog> is assignable to Array<Animal>
    const animals: Animal[] = dogs;

    // Unsafe mutation!
    // We push a generic Animal into the array. The array is technically 'dogs'.
    animals.push(new Animal());

    // Runtime Error potential:
    // The second element is an Animal, but 'dogs' thinks it contains only Dogs.
    // dogs[1].bark(); // Error: bark is undefined on the generic Animal.
    console.log("Unsafe mutation complete. accessing dogs[1].bark() would crash.");
}

main();
```

Hours 3-4 Focus on Delegate Variance

C# Example

```
using System;

class Program {
    // Contravariance: Delegate expects 'string' (Specific), we give 'object' (Generic).
    // "I need a method that can handle a string."
    delegate void StringHandler(string s);

    // "I can handle any object."
    static void HandleObject(object o) {
        Console.WriteLine($"Handled object: {o}");
    }

    static void Main() {
        // Assignment works:
        // A method taking 'object' can definitely handle 'string'.
        StringHandler handler = HandleObject; // Safe Contravariance

        handler("Hello");
    }
}
```

TypeScript Example

```
// REQUIREMENT: Ensure "strictFunctionTypes": true in tsconfig.json
// Without this, TypeScript allows unsafe Bivariance.

class Animal { name: string = "Animal"; }
class Dog extends Animal { bark() { console.log("Woof"); } }

type DogHandler = (d: Dog) => void;

function main() {
    // Contravariance: We use a handler that accepts a WIDER type (Animal)
    // to satisfy a requirement for a NARROWER type (Dog).
    const handleAnimal = (a: Animal) => {
        console.log(`Handling ${a.name}`);
    };

    // Safe assignment:
    // DogHandler expects a function that can take a Dog.
    // handleAnimal takes any Animal. Since Dog IS-A Animal, this is safe.
    const specificHandler: DogHandler = handleAnimal;

    specificHandler(new Dog());
}

main();
```

Day 5 Unify Holistic Metrics including Maintainability & Halstead

Hours 1-2 Calculate Halstead Complexity Measures & Modern Critique

Halstead Metrics (1977) treat code as a "bag of tokens," ignoring structure. A concise, 10-line functional pipeline might have the same Volume as a tangled 10-line loop. However, since the Maintainability Index relies on Halstead, we must understand it.

C# Example

```
class Program {
    // POOR: High Volume (Verbose)
    static void VerboseCalc() {
        // Ops (3): "int", "=", ";"
        // Opds (2): "a", "3"
        int a = 3;
        int b = 4;
        // Intermediate variables inflate Operator/Operand counts
        int temp1 = a * a;
        int temp2 = b * b;
        int sum = temp1 + temp2;
        double c = Math.Sqrt(sum);
    }

    // GOOD: Low Volume (Concise)
    static void ConciseCalc() {
        // Fewer tokens = Lower Volume
        double c = Math.Sqrt((3 * 3) + (4 * 4));
    }

    static void Main() {
```

```
        VerboseCalc();
        ConciseCalc();
    }
}
```

TypeScript Example

```
// POOR: High Volume
function verboseCalc(): void {
    const a = 3;
    const b = 4;
    // "const", "sqA", "=", "a", "*", "a", ";" -> Many tokens
    const sqA = a * a;
    const sqB = b * b;
    const sum = sqA + sqB;
    const c = Math.sqrt(sum);
}

// GOOD: Low Volume
function conciseCalc(): void {
    const c = Math.sqrt(3**2 + 4**2);
}

// Usage
verboseCalc();
conciseCalc();
```

Hours 3-4 Compute The Maintainability Index (MI)

It is important to note the nuance between the two versions of MI.

- **Classic MI:** Ranges from 0 to 171.
- **Microsoft MI:** Ranges from 0 to 100 (Logarithmic). Always specify which scale you are using in your reports.

C# Example

```
class OrderProcessor {
    // POOR: Low MI (< 65)
    // High CC + High Volume + High LOC
    public void ProcessBad(Order order) {
        if (order != null) {
            if (order.Items != null) {
                foreach (var item in order.Items) {
                    if (item == "Electronics") {
                        // ... Deep nesting ...
                    }
                }
            }
        }
    }

    // GOOD: High MI (> 85)
    // Flat structure + Concise Logic
    public void ProcessGood(Order order) {
        if (order?.Items == null) return;

        var electronics = order.Items.Where(i => i == "Electronics");
        foreach (var item in electronics) {
            // ... Linear flow ...
        }
    }
}

class Program {
    static void Main() {
        var op = new OrderProcessor();
        op.ProcessBad(new Order());
        op.ProcessGood(new Order());
    }
}
```

TypeScript Example

```
class OrderProcessor {
    // POOR: Low MI
    processBad(order: Order | null): void {
        if (order) {
            if (order.items) {
                // ... Deep nesting ...
```

```
            }
        }
    }

    // GOOD: High MI
    processGood(order: Order | null): void {
        if (!order?.items?.length) return;

        order.items
            .filter(item => item === "Electronics")
            .forEach(item => {
                // ... Functional pipeline ...
            });
    }
}

// Usage
const op = new OrderProcessor();
op.processBad({ items: [] });
op.processGood({ items: [] });
```