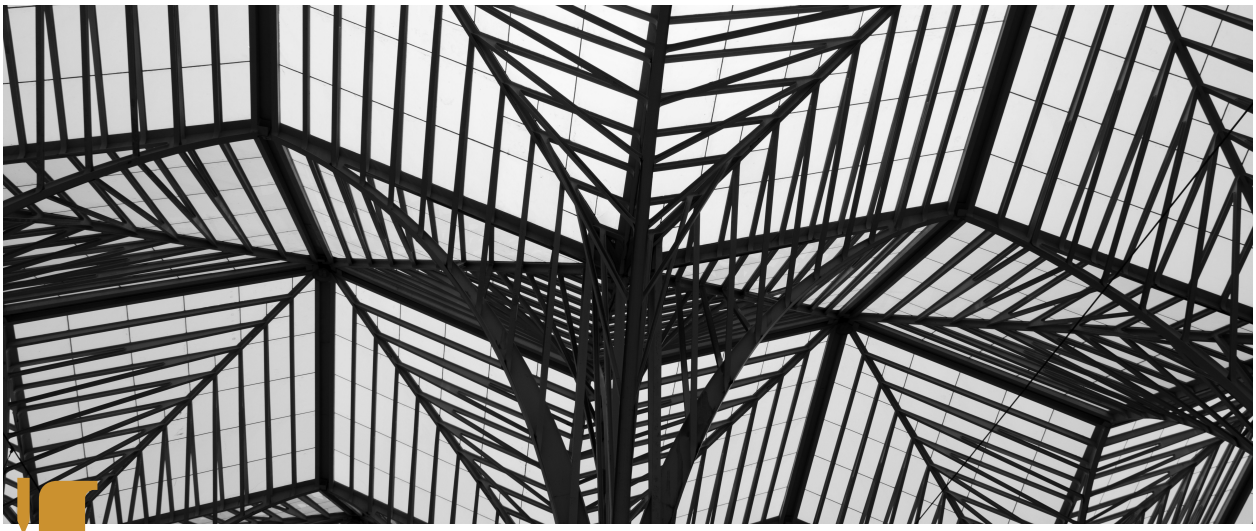


Cyclomatic complexity

▼ Introduction

Cyclomatic complexity (CC) is a software metric that measures the complexity of a program. It counts the number of independent paths through the code, determined by the number of control structures such as `if-else`, `for`, `switch`.

▼ How to calculate

Cyclomatic Complexity measures the number of **linearly independent paths** in a program. Each independent path represents a decision point in the code.

- **Key Points:**
  - It's calculated based on the control flow graph of the program.
  - CC gives an idea of how complex and interconnected your code is.

▼ Formula

Formula: $CC = E - N + 2P$

Where:

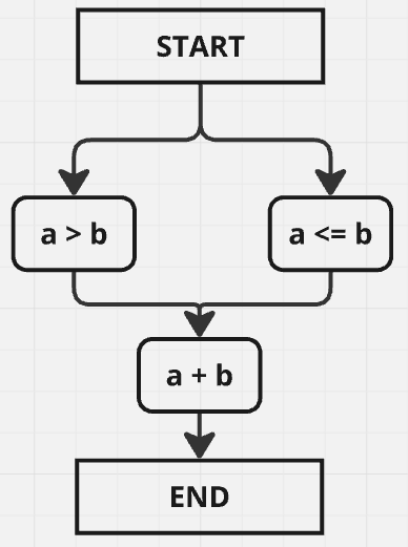
- E — number of edges in the control graph.
- N — number of nodes in the control graph.
- P — number of components (usually 1 for a single function). (number of ways to exit or end the program),

▼ Example 1

```
def example(a, b):
    if a > b:
        print("a is greater")
    else:
        print("b is greater or equal")
    return a + b
```

Program control graph for the program has 5 edges and 5 nodes.

Hence: $CC = 5 - 5 + 2 * 1 = 2$



Program control graph

In this example:

- Control structures: `if-else` (1 decision point).
- CC=2: One path for `if`, another for `else`.

▼ Example 2

```
def example(a, b):
    if a > b:
        print("a > b")
    elif a == b:
        print("a == b")
    else:
        print("a < b")

    for i in range(a):
```

```
        print(i)
    return b
```

In this example:

- `if-elif-else`: 2 branches. `for` loop: Adds one additional path.
- CC=4: 3 decision branches + 1 loop.

Impact of high cyclomatic complexity

Difficulties

High cyclomatic complexity may lead to following difficulties:

1. **Difficult testing:** higher CC means more tests are required to cover all paths.
2. **Increased bug probability:** complex code is more bug-prone.
3. **Reduces readability:** code with high CC becomes less intuitive.

Thresholds

Thresholds are relative to each program individually.

General thresholds:

- CC ≤ 10: acceptable, manageable complexity.
- CC < 10: consider refactoring.
- CC > 20: high complexity — urgent refactoring.

Techniques to minimize cyclomatic complexity

1. Brake down functions (refactoring):

Brake down functions: Large, monolithic functions tend to have higher CC. Splitting them into small, focused functions reduces complexity and improves readability.

Example:

```
# Complex function with CC = 5
def calculate_total(items, discount, tax):
    if discount > 0:
        items = apply_discount(items, discount)
    total = 0
    for item in items:
        if item.price > 100:
            tax_rate = 0.1
        else:
            tax_rate = 0.05
        total += item.price + (item.price * tax_rate)
```

```
    return total
```

Refactored:

```
def calculate_total(items, discount, tax):
    items = apply_discount(items, discount) if discount > 0 \
    else items
    return sum(apply_tax(item) for item in items)

def apply_tax(item):
    tax_rate = 0.1 if item.price > 100 else 0.05
    return item.price + (item.price * tax_rate)
```

New CC: 2 for `calculate_total` + 1 for `apply_tax` = 3.

▼ 2. Use Guard Clauses:

Use guard clauses: Guard clauses help eliminate deeply nested conditions by returning early.

Before:

```
def process_order(order):
    if order.is_valid():
        if order.is_paid:
            ship_order(order)
        else:
            print("Order not paid")
    else:
        print("Invalid order")
```

After:

```
def process_order(order):
    if not order.is_valid():
        print("Invalid order")
        return

    if not order.is_paid:
        print("Order not paid")
        return
```

```
    ship_order(order)
```

Result: Reduced nesting and CC.

▼ 3. Replace conditions with Polymorphism:

Replace conditions with Polymorphism: Excessive `if-else` or `switch` statements can be replaced with polymorphism or design patterns like *Strategy*.

Example:

```
# High CC due to multiple conditions
def process_payment(payment_type, amount):
    if payment_type == "credit_card":
        process_credit_card(amount)
    elif payment_type == "paypal":
        process_paypal(amount)
    elif payment_type == "bank_transfer":
        process_bank_transfer(amount)
    else:
        raise ValueError("Unsupported payment type")
```

Refactored with Polymorphism:

```
class PaymentProcessor:
    def process(self, amount):
        raise NotImplementedError()

class CreditCardPayment(PaymentProcessor):
    def process(self, amount):
        process_credit_card(amount)

class PayPalPayment(PaymentProcessor):
    def process(self, amount):
        process_paypal(amount)

class BankTransferPayment(PaymentProcessor):
    def process(self, amount):
        process_bank_transfer(amount)

# Usage
```

```
payment = PayPalPayment()
payment.process(amount)
```

Effect: CC reduced to the base class.

▼ 4. Use ternary operators:

Use ternary operators: For simple conditions, use concise expressions instead of full `if-else` blocks.

Example:

```
# Before
if a > b:
    max_value = a
else:
    max_value = b

# After
max_value = a if a > b else b
```

▼ 5. Adopt functional programming principles:

Adopt functional programming principles: Functional programming encourages immutability and pure functions, which tend to have low CC.

Example: Use **map**, **filter**, or **reduce** instead of loops and conditions.