Tentu, berikut adalah dokumentasi lengkap dari kode ArithmeticOperations.cs, yang menjelaskan fungsionalitas dan alasan desainnya baris per baris (atau lebih tepatnya, blok per blok fungsional).

Dokumentasi: ArithmeticOperations.cs

Dokumen ini merinci fungsionalitas, kegunaan, dan alasan di balik setiap bagian dari file ArithmeticOperations.cs.

Ringkasan File

- **Namespace:** MiniPhotoshop.Logic.ImageProcessing
- **Kelas:** ArithmeticOperations
- **Tujuan:** Kelas ini adalah *utility class* (kelas pembantu) statis yang menyediakan metode untuk melakukan empat operasi aritmatika dasar (Tambah, Kurang, Kali, Bagi) pada dua buah gambar (System.Drawing.Bitmap).
- **Fitur Utama:** Fitur desain yang paling penting adalah kemampuannya untuk menangani dua gambar yang memiliki **dimensi berbeda**. Kanvas hasil akan memiliki ukuran lebar maksimum x tinggi maksimum dari kedua gambar tersebut.

Analisis Kode per Blok

Blok 1: Definisi Kelas dan Namespace

C#

```
using System;
```

```
using System.Drawing;

namespace MiniPhotoshop.Logic.ImageProcessing
{
    /// <summary>
    /// Operasi aritmatika dua citra dengan kanvas hasil sebesar
    /// dimensi maksimum di antara keduanya.
    /// </summary>
    public static class ArithmeticOperations
    {
```

- **using System; / using System.Drawing;**
  - **Penjelasan:** Mengimpor *namespace* dasar dari .NET.
  - **Fungsi:** System diperlukan untuk Func<> (delegasi). System.Drawing diperlukan untuk semua kelas terkait gambar seperti Bitmap dan Color.
  - **Alasan:** Tanpa ini, kode tidak akan mengenali tipe data Bitmap, Color, atau Func.
- **public static class ArithmeticOperations**
  - **Penjelasan:** Mendeklarasikan kelas *utility* yang bersifat public dan static.
  - **Fungsi:**
    - public: Membuat kelas ini dapat diakses dari bagian lain aplikasi.
    - static: Menandakan bahwa kelas ini tidak dapat diinstansiasi (Anda tidak bisa melakukan new ArithmeticOperations()). Semua metodenya dipanggil langsung dari nama kelas (misal, ArithmeticOperations.Add(...)).
  - **Alasan:** Desain ini ideal untuk *helper class* yang hanya berisi metode fungsional dan tidak perlu menyimpan *state* atau data internal apa pun.

Blok 2: Metode Helper Clamp

C#

```
        // Menjepit nilai channel ke rentang valid 0..255
        private static int Clamp(int value)
        {
            if (value < 0) return 0;
            if (value > 255) return 255;
            return value;
        }
```

- **private static int Clamp(int value)**
  - **Penjelasan:** Ini adalah metode pembantu internal yang "menjepit" sebuah nilai integer agar selalu berada dalam rentang 0 hingga 255.
  - **Fungsi:** Mencegah nilai *channel* warna (Merah, Hijau, Biru) keluar dari rentang yang valid.
  - **Alasan:** Sebuah *channel* warna disimpan sebagai byte (8-bit), yang hanya bisa menampung nilai 0 (tidak ada warna) hingga 255 (warna penuh).
    - **Contoh:** Operasi Add (Penjumlahan) 200 + 100 akan menghasilkan 300. Clamp akan memaksanya menjadi 255 (putih).
    - **Contoh:** Operasi Subtract (Pengurangan) 50 - 80 akan menghasilkan -30. Clamp akan memaksanya menjadi 0 (hitam).
  - Tanpa Clamp, nilai yang tidak valid ini akan menyebabkan *exception* (kesalahan) saat mencoba membuat Color baru, atau akan "meluap" (integer overflow) dan menghasilkan warna yang salah kaprah.

Blok 3: Mesin Operasi Utama PerformOperation

C#

```
    // Mesin umum: loop piksel dan terapkan fungsi operasi (Add/Sub/Mul/Div)
    private static Bitmap PerformOperation(
      Bitmap imgA, Bitmap imgB, Func<Color, Color, Color> operation)
    {
        // [Baris 24-26]
        int newWidth = Math.Max(imgA.Width, imgB.Width);
        int newHeight = Math.Max(imgA.Height, imgB.Height);
        Bitmap resultBmp = new Bitmap(newWidth, newHeight);

        // [Baris 28-56]
        for (int y = 0; y < newHeight; y++)
        {
            for (int x = 0; x < newWidth; x++)
            {
                bool inA = (x < imgA.Width && y < imgA.Height);
                bool inB = (x < imgB.Width && y < imgB.Height);
```

```
            if (inA && inB)
            {
                Color cA = imgA.GetPixel(x, y);
                Color cB = imgB.GetPixel(x, y);
                resultBmp.SetPixel(x, y, operation(cA, cB));
            }
            else if (inA)
            {
                resultBmp.SetPixel(x, y, imgA.GetPixel(x, y));
            }
            else if (inB)
            {
                resultBmp.SetPixel(x, y, imgB.GetPixel(x, y));
            }
            else
            {
                resultBmp.SetPixel(x, y, Color.Black);
            }
        }
    }
    return resultBmp;
}
```

- **private static Bitmap PerformOperation(...)**
  - **Penjelasan:** Ini adalah metode inti yang melakukan semua pekerjaan berat. Metode ini mengabstraksi logika perulangan (looping) piksel.
  - **Fungsi:** Menerima dua gambar (A dan B) dan sebuah "fungsi operasi" (Func<Color, Color, Color> operation). Fungsi ini adalah parameter yang akan memberi tahu metode ini *apa* yang harus dilakukan (menambah, mengurangi, dll.) pada dua piksel yang sedang diproses.
  - **Alasan:** Ini adalah contoh bagus dari **Prinsip DRY (Don't Repeat Yourself)**. Daripada menulis logika *looping* for (x...) dan for (y...) yang sama persis di dalam Add, Subtract, Multiply, dan Divide, kita menulisnya *satu kali* di sini. Metode publik (Add, Subtract, dll.) hanya perlu menyediakan fungsi operasinya saja. Ini membuat kode lebih bersih dan mudah dipelihara.
- **int newWidth = Math.Max(...) / int newHeight = Math.Max(...)**
  - **Penjelasan:** Menghitung dimensi untuk kanvas hasil.
  - **Fungsi:** Mengambil nilai *terbesar* dari lebar kedua gambar, dan nilai *terbesar* dari tinggi kedua gambar.
  - **Alasan:** Sesuai ringkasan di atas kelas, ini memastikan bahwa kanvas baru cukup besar untuk menampung *seluruh* area dari kedua gambar, mencegah pemotongan (clipping).
- **bool inA = ... / bool inB = ...**

    - **Penjelasan:** Dua variabel boolean untuk memeriksa apakah koordinat (x, y) saat ini berada di dalam batas gambar A atau gambar B.
    - **Alasan:** Ini adalah inti dari penanganan gambar berukuran beda.
- **Blok if (inA && inB)**
    - **Penjelasan:** Jika piksel (x, y) ada di *kedua* gambar.
    - **Fungsi:** Ambil warna dari kedua piksel (cA dan cB), lalu jalankan operation (misalnya, fungsi tambah) pada keduanya, dan simpan hasilnya di gambar baru.
    - **Alasan:** Ini adalah skenario utama di mana operasi aritmatika (blending) terjadi.
- **Blok else if (inA) / else if (inB)**
    - **Penjelasan:** Jika piksel (x, y) hanya ada di gambar A (karena gambar B lebih kecil), atau sebaliknya.
    - **Fungsi:** Cukup salin piksel dari gambar yang lebih besar ke kanvas hasil.
    - **Alasan:** Ini memastikan bahwa bagian gambar yang "tidak memiliki pasangan" tidak hilang atau terpotong, melainkan tetap ditampilkan di hasil akhir.
- **Blok else**
    - **Penjelasan:** Jika piksel tidak ada di A maupun B.
    - **Fungsi:** Mengisi area tersebut dengan Color.Black (hitam).
    - **Alasan:** Area ini adalah area "mati" yang tercipta jika satu gambar sangat lebar dan gambar lain sangat tinggi. Ini adalah pengisi *default* yang aman.

Blok 4: Metode Publik Add

C#

```
    // Tambah (channel-wise) dengan clamp
    public static Bitmap Add(Bitmap imgA, Bitmap imgB)
    {
        return PerformOperation(imgA, imgB, (cA, cB) =>
        {
            int r = Clamp(cA.R + cB.R);
            int g = Clamp(cA.G + cB.G);
            int b = Clamp(cA.B + cB.B);
            return Color.FromArgb(r, g, b);
        });
    }
```

- **public static Bitmap Add(...)**

- **Penjelasan:** Metode publik untuk menjumlahkan dua gambar.
- **Fungsi:** Memanggil PerformOperation dan memberikannya *lambda expression* ((cA, cB) => ...) sebagai parameter operation.
- **Alasan:** *Lambda expression* ini mendefinisikan *apa* arti "Add":
  1. Ambil channel Merah dari A dan B, jumlahkan: cA.R + cB.R.
  2. Amankan hasilnya dengan Clamp().
  3. Ulangi untuk Hijau (G) dan Biru (B).
  4. Kembalikan Color baru dari nilai R, G, B yang sudah dijumlahkan.

Blok 5: Metode Publik Subtract

C#

```
    // Kurang (channel-wise) dengan clamp
    public static Bitmap Subtract(Bitmap imgA, Bitmap imgB)
    {
        return PerformOperation(imgA, imgB, (cA, cB) =>
        {
            int r = Clamp(cA.R - cB.R);
            int g = Clamp(cA.G - cB.G);
            int b = Clamp(cA.B - cB.B);
            return Color.FromArgb(r, g, b);
        });
    }
```

- **public static Bitmap Subtract(...)**
  - **Penjelasan:** Metode publik untuk mengurangkan gambar B dari gambar A.
  - **Fungsi:** Serupa dengan Add, ini memanggil PerformOperation dengan lambda yang berbeda.
  - **Alasan:** Lambda ini mendefinisikan "Subtract": int r = Clamp(cA.R - cB.R). Ini berguna untuk mencari perbedaan (differencing) antar gambar.

Blok 6: Metode Publik Multiply

C#

```
    // Kali (ternormalisasi 0..1 lalu dikembalikan ke 0..255)
    public static Bitmap Multiply(Bitmap imgA, Bitmap imgB)
    {
        return PerformOperation(imgA, imgB, (cA, cB) =>
        {
            int r = Clamp((int)((cA.R / 255.0) * (cB.R / 255.0) * 255.0));
            int g = Clamp((int)((cA.G / 255.0) * (cB.G / 255.0) * 255.0));
            int b = Clamp((int)((cA.B / 255.0) * (cB.B / 255.0) * 255.0));
            return Color.FromArgb(r, g, b);
        });
    }
```

- **public static Bitmap Multiply(...)**
  - **Penjelasan:** Metode publik untuk mengalikan dua gambar, yang dikenal sebagai *blend mode* "Multiply".
  - **Fungsi:** Mengimplementasikan perkalian *blend mode* dengan **normalisasi**.
  - **Alasan (Normalisasi):** Kita tidak bisa langsung mengalikan 150 * 150, karena hasilnya 22500 (jauh di luar 255).
    1. cA.R / 255.0: Mengubah nilai 0-255 menjadi rentang 0.0-1.0 (normalisasi).
    2. (...) * (...): Mengalikan nilai yang sudah dinormalisasi (misal, 0.6 * 0.6 = 0.36).
    3. ... * 255.0: Mengembalikan hasil 0.0-1.0 tadi ke rentang 0-255 (misal, 0.36 * 255.0 = 91.8).
    4. (int)...: Mengubahnya menjadi integer (91) dan Clamp mengamankannya (meskipun dalam *multiply* tidak akan pernah *overflow*).
  - Ini adalah cara standar dan matematis yang benar untuk mengimplementasikan *blend mode* Multiply, yang hasilnya selalu lebih gelap.

Blok 7: Metode Publik Divide

C#

```
    // Bagi (ternormalisasi) dengan proteksi divide-by-zero
```

```
    public static Bitmap Divide(Bitmap imgA, Bitmap imgB)
    {
        return PerformOperation(imgA, imgB, (cA, cB) =>
        {
            double divR = (cB.R == 0) ? 1 : (cB.R / 255.0);
            double divG = (cB.G == 0) ? 1 : (cB.G / 255.0);
            double divB = (cB.B == 0) ? 1 : (cB.B / 255.0);

            int r = Clamp((int)((cA.R / 255.0) / divR * 255.0));
            int g = Clamp((int)((cA.G / 255.0) / divG * 255.0));
            int b = Clamp((int)((cA.B / 255.0) / divB * 255.0));
            return Color.FromArgb(r, g, b);
        });
    }
```

- **public static Bitmap Divide(...)**
  - **Penjelasan:** Metode publik untuk membagi gambar A dengan gambar B.
  - **Fungsi:** Mengimplementasikan *blend mode* Divide dengan normalisasi dan proteksi kesalahan.
  - **Alasan (Proteksi *Divide-by-Zero*):**
    1. double divR = (cB.R == 0) ? 1 : (cB.R / 255.0);
       - Ini adalah bagian paling kritis. Kita tidak bisa membagi dengan nol.
       - **Penjelasan:** "Jika channel Merah di B adalah 0, anggap nilai pembaginya adalah 1. Jika tidak, gunakan nilai normalisasinya (cB.R / 255.0)."
       - **Alasan:** Ini mencegah *error* DivideByZeroException total. Membagi dengan 1 berarti (cA.R / 255.0) / 1.0, yang hasilnya sama dengan cA.R (setelah normalisasi), ini adalah perilaku yang aman dan logis.
    2. Logika (cA.R / 255.0) / divR * 255.0 sama seperti Multiply, yaitu melakukan matematika pada rentang 0.0-1.0 yang ternormalisasi. Hasilnya akan selalu lebih cerah.