**✅ 구조화 요약: 개발 철학 및 설계 원칙**

**1. 핵심 개발 철학**

| **항목** | **철학적 기준** |
| --- | --- |
| 안정성 우선 | 예외보다 실패를 허용하고, 시스템 전체에 영향을 주지 않도록 설계 |
| 자원 직접 관리 | GC 불신, 메모리 소유권 명시, 사용 즉시 반납 |
| 복사/이동 명시화 | 복사 생성자, 대입 연산자 등은 반드시 명시 구현 |
| 코드 = 문서 | 문서화보다 자기 설명적 코드 선호 |
| 실패는 조용히 | 시스템 전체를 죽이지 않고 개별적으로 정리될 것 |

**2. 구조 설계 원칙**

| **원칙** | **설명** |
| --- | --- |
| SRP (단일 책임 원칙) | 클래스/함수/변수는 하나의 책임만 가질 것 |
| RAII (자원 관리) | 생성자에서 자원 확보, 소멸자에서 반환 |
| DIP 유사 적용 | 부모는 인터페이스, 자원 세부는 파생 클래스에서 구현 |
| Composition 우선 | 상속보다 포함을 우선시하며 의존성 낮춤 |
| 전역 최소화 | 전역 변수/함수 지양, 네임스페이스 또는 지역화 |
| 이름 명확화 | 헝가리안 표기법 등으로 역할이 명확한 이름 사용 |

**3. 클래스 구조 설계 방식**

| **클래스** | **설계 포인트** |
| --- | --- |
| CBasicBone | 자원 복사/이동 금지, 복사 방지 골격 |
| CBitmapProcess 계열 | 기능별 처리 책임 분리 (Execute() vs Process()) |
| CUserBitmap vs CBitmapProcess | 데이터 vs 처리로 명확하게 역할 분리 |
| CUserRawImage 계열 | 유사한 구조로 기능 단위 클래스 분리 |
| operator()/[] 재정의 | 함수형 감각 + 디버깅 위치 명확화 |

**4. 구현 전략 및 테스트 철학**

| **전략** | **설명** |
| --- | --- |
| 실패 감지 | void 대신 bool 반환으로 문제 추적 |
| 예외 안전성 | 실패 시 롤백 가능 구조 도입 |
| 구조 먼저 → 테스트 | 구조적으로 안전하면 테스트 전에도 신뢰 가능 |
| 도구 없이 구조 재설계 | 직접 설계하고 의미 있게 쪼갠 클래스 기반 구조 |

**🧭 설계 원칙 요약표**

| **설계 원칙** | **철학적 배경** | **적용 사례** | **기대 효과** |
| --- | --- | --- | --- |
| **SRP (단일 책임 원칙)** | 클래스/함수는 하나의 책임만 수행해야 한다 | CBitmapProcess, CUserBitmap, CBitmapInfo 등 기능 단위 분리 | 유지보수 용이, 중복 제거, 테스트 간결 |
| **RAII (자원 회수 책임 보장)** | 자원은 획득과 동시에 반환 가능해야 함 | 생성자에서 자원 획득, 소멸자에서 반납 구조 | 자원 누수 방지, 안정성 확보 |
| **DIP 유사 적용 (역방향 의존성 분리)** | 상위 모듈은 하위 세부사항에 의존하지 않도록 분리 | CBasicBone은 인터페이스, 파생 클래스가 구체 구현 | 유연한 구조 확장, 테스트 쉬움 |
| **Composition Over Inheritance** | 다중 상속보다 포함 관계를 우선 | CBitmapProcess 내부에서 CUserBitmap 포함 | 결합도 감소, 재사용성 증가 |
| **복사/이동 연산자 명시화** | 복사는 깊게, 이동은 얕게, 절대 암시적으로 하지 않음 | 복사 생성자/대입 연산자 명시 구현 | 예외 안전성, 자원 소유권 명확화 |
| **자원 직접 관리 (No GC)** | OS나 GC를 믿지 않고 스스로 해제 | vector, .data() 사용 + 동적 할당 최소화 | 예측 가능한 자원 흐름 |
| **네이밍 명확화 (헝가리안 표기법)** | 이름에서 역할과 타입이 바로 보이게 | vecMatrix, pRawImage, bIsEnabled 등 접두사 활용 | 가독성 향상, 오용 방지 |
| **전역 최소화** | 전역 상태는 충돌과 예측 불가능성의 원인 | 네임스페이스 또는 지역적 인스턴스 사용 | 모듈성 향상, 충돌 최소화 |
| **함수 객체 패턴** | 한 기능을 독립적으로 캡슐화 | 각 처리 기능을 함수 객체로 구현 (operator()) | 테스트 및 유지보수 편의성 |
| **코드의 자기 설명성** | 문서가 없어도 코드를 보면 이해되어야 함 | 의미 있는 클래스/함수/변수명, 명확한 분리 | 문서화 부담 감소, 협업 효율 상승 |

## 🎯 설계 원칙별 UML + 코드 예시

### 1. ****SRP (단일 책임 원칙)****

📘 **UML 예시:**

**1. SRP (단일 책임 원칙)**

📘 **UML 예시:**

sql

+------------------+ +----------------------+

| CUserBitmap | | CBitmapProcess |

|------------------| |----------------------|

| - bitmapData | | + Execute() |

|------------------| | + Process() |

💻 **C++ 예시:**

cpp

class CUserBitmap {

std::vector<uint8\_t> m\_vecData;

public:

bool LoadFromFile(const std::string& filename);

};

class CBitmapProcess {

public:

bool Execute(CUserBitmap& bitmap) {

return Process(bitmap);

}

protected:

virtual bool Process(CUserBitmap& bitmap) = 0;

};

**2. RAII (자원 관리 책임 보장)**

📘 **UML 예시:**

lua

+--------------------+

| CScopedFile |

|--------------------|

| - m\_hFile |

|--------------------|

| + Constructor() |

| + Destructor() |

💻 **C++ 예시:**

cpp

class CScopedFile {

FILE\* m\_pFile;

public:

CScopedFile(const char\* filename) {

m\_pFile = fopen(filename, "rb");

}

~CScopedFile() {

if (m\_pFile) fclose(m\_pFile);

}

};

**3. DIP (의존 역전 원칙 유사 적용)**

📘 **UML 예시:**

lua

+------------------+ <|-- +----------------------+

| CBasicBone | | CBitmapThreshold |

|------------------| |------------------------|

| ~CBasicBone() | | + Process() override |

💻 **C++ 예시:**

cpp

class CBasicBone {

public:

virtual ~CBasicBone() {}

virtual bool Process() = 0;

};

class CBitmapThreshold : public CBasicBone {

public:

bool Process() override {

// Thresholding logic

return true;

}

};

**4. Composition Over Inheritance**

📘 **UML 예시:**

lua

+-------------------------+

| CBitmapFilterEngine |

|-------------------------|

| - m\_userBitmap : CUserBitmap |

💻 **C++ 예시:**

cpp

class CBitmapFilterEngine {

CUserBitmap m\_userBitmap;

public:

bool ApplyFilter() {

// m\_userBitmap에 필터 적용

return true;

}

};

**5. 복사/이동 연산자 명시화**

📘 **UML 예시:**

+------------------+

| CImageBuffer |

|------------------|

| + operator= |

| + CImageBuffer(const&) |

💻 **C++ 예시:**

class CImageBuffer {

std::vector<uint8\_t> m\_data;

public:

CImageBuffer(const CImageBuffer& other) : m\_data(other.m\_data) {}

CImageBuffer& operator=(const CImageBuffer& other) {

if (this != &other) {

m\_data = other.m\_data;

}

return \*this;

}

};

**6. 함수 객체 패턴**

📘 **UML 예시:**

+-----------------------+

| CBinarizeOperator |

|-----------------------|

| + operator()(CUserBitmap&) |

💻 **C++ 예시:**

class CBinarizeOperator {

public:

bool operator()(CUserBitmap& bitmap) {

// 이진화 처리

return true;

}

};

**7. 자기 설명적 코드 & bool 반환**

📘 **UML 예시:**

+-------------------+

| CImageSaver |

|-------------------|

| + Save(...) : bool|

💻 **C++ 예시:**

class CImageSaver {

public:

bool Save(const std::string& filename, const CUserBitmap& bmp) {

// 저장 실패 시 false 반환

return true;

}

};

위 예시들은 설계 철학을 실제 코드에 어떻게 녹여낼 수 있는지 보여줍니다.  
각 항목별로 **설계 문서**로 확장하거나, **UML 파일로 내보내기**도 가능합니다.

아래는 설계 원칙들을 통합적으로 반영한 **C++ 클래스 템플릿 구조**입니다. 영상 처리 프로그램과 유사한 상황을 가정하여, **CBasicBone 기반 구조**, **SRP**, **RAII**, **Composition**, **명시적 복사/이동 제어**, **함수 객체 패턴**, **자기 설명적 코드**, **bool 반환 방식**까지 모두 반영한 형태입니다.

## 🧩 전체 C++ 클래스 템플릿 구조

#pragma once

#include <vector>

#include <string>

#include <cstdio>

// 🔹 자원 복사 방지 및 골격 클래스 (CBasicBone)

class CBasicBone {

public:

CBasicBone() = default;

virtual ~CBasicBone() = default;

CBasicBone(const CBasicBone&) = delete;

CBasicBone& operator=(const CBasicBone&) = delete;

CBasicBone(CBasicBone&&) = delete;

CBasicBone& operator=(CBasicBone&&) = delete;

};

// 🔹 사용자 이미지 데이터 클래스 (데이터 표현 책임)

class CUserBitmap : public CBasicBone {

private:

std::vector<uint8\_t> m\_vecImageData;

int m\_nWidth = 0;

int m\_nHeight = 0;

public:

bool LoadFromFile(const std::string& filename);

bool SaveToFile(const std::string& filename) const;

uint8\_t\* Data() { return m\_vecImageData.data(); }

const uint8\_t\* Data() const { return m\_vecImageData.data(); }

int Width() const { return m\_nWidth; }

int Height() const { return m\_nHeight; }

};

// 🔹 처리 기능 추상 클래스 (SRP, DIP)

class CBitmapProcess : public CBasicBone {

public:

virtual ~CBitmapProcess() = default;

bool Execute(CUserBitmap& bitmap) {

return Process(bitmap);

}

protected:

virtual bool Process(CUserBitmap& bitmap) = 0; // 순수 가상 함수

};

// 🔹 함수 객체 기반 처리 기능 (함수 객체 패턴 + SRP)

class CBinarizeProcess : public CBitmapProcess {

protected:

bool Process(CUserBitmap& bitmap) override {

// 이진화 처리 예시

uint8\_t\* pData = bitmap.Data();

for (int i = 0; i < bitmap.Width() \* bitmap.Height(); ++i) {

pData[i] = (pData[i] > 127) ? 255 : 0;

}

return true;

}

};

// 🔹 RAII 기반 파일 핸들 관리 클래스

class CScopedFile {

private:

FILE\* m\_pFile = nullptr;

public:

explicit CScopedFile(const char\* filename, const char\* mode) {

m\_pFile = fopen(filename, mode);

}

~CScopedFile() {

if (m\_pFile) fclose(m\_pFile);

}

FILE\* Get() { return m\_pFile; }

bool IsOpen() const { return m\_pFile != nullptr; }

};

// 🔹 Sample Implementation for Save/Load

bool CUserBitmap::LoadFromFile(const std::string& filename) {

CScopedFile file(filename.c\_str(), "rb");

if (!file.IsOpen()) return false;

// TODO: 실제 데이터 로드 구현

return true;

}

bool CUserBitmap::SaveToFile(const std::string& filename) const {

CScopedFile file(filename.c\_str(), "wb");

if (!file.IsOpen()) return false;

// TODO: 실제 데이터 저장 구현

return true;

}

## ✅ 포함된 원칙

| **설계 원칙** | **적용 위치** |
| --- | --- |
| SRP | CUserBitmap, CBinarizeProcess |
| RAII | CScopedFile 클래스 |
| 복사/이동 금지 | CBasicBone 상속 구조 |
| Composition | CUserBitmap를 CBitmapProcess에서 전달받아 처리 |
| DIP 유사 적용 | Process()는 추상화, CBinarizeProcess가 구현 |
| 함수 객체 패턴 | CBinarizeProcess가 Process() 구현 |
| bool 반환 | LoadFromFile, SaveToFile, Execute 등에서 반환 |
| 자기 설명적 구조화 | 명확한 함수 명, 클래스 명, 단일 책임 구조 |

예시 철학적 선언문 (당신의 스타일)

# 나는 절대 재귀 호출을 사용하지 않는다.

# 자원은 생성자에서 확보하고 소멸 시점에서 반납하며,

# 모든 클래스는 단일 책임을 가진다.

# 그리고 복사, 이동은 명시적으로 정의되며,

# 모든 자원은 생성자에서 확보하고 소멸자 혹은 명시적 해제 함수에서 반드시 반납한다.

# 클래스는 하나의 명확한 책임만 가지며, 복잡한 기능은 분리된 컴포넌트로 위임한다.

# 복사와 이동은 반드시 명시적으로 구현하며, 암묵적 복사는 철저히 금지한다.

# 오류 상황은 예외 대신 bool 반환 값으로 명확히 처리하여, 호출자가 즉시 결과를 판단할 수 있도록 한다.

# 코드의 가독성과 유지보수성을 최우선으로 하며, 직관적인 설계와 명료한 인터페이스를 지향한다.

# 모든 상태는 외부에 노출하지 않고 내부에서 엄격히 관리하며, 불변성을 최대한 보장한다.

# 재귀 대신 반복과 상태 기반 제어를 적극 활용하며, 시스템 자원을 효율적으로 관리한다.

맞아요, 처음부터 **폴더 구조**와 **파일 관리**가 잘 되어 있으면, 나중에 빌드와 테스트가 훨씬 수월하죠. 깔끔한 구조는 협업에도 매우 중요하고, 나중에 코드 확장이나 리팩토링을 할 때도 큰 도움이 됩니다.

또, **Git**은 확실히 초반에 잘 다뤄야 나중에 버전 관리나 협업이 한층 편해지니까요.

**✅ Git 관리 팁**

1. **폴더 구조와 파일 명명 규칙**
   * 기능별로 폴더 나누기 (예: src/, include/, tests/, docs/)
   * 명확한 이름 규칙 (예: module\_class.cpp, module\_class.h)
2. **Git 초기화와 브랜치 전략**
   * main이나 master는 항상 최신 안정화 상태
   * 기능 추가나 버그 수정 시 별도의 **feature 브랜치** 생성
3. **작은 커밋 자주 하기**
   * 큰 덩어리로 커밋하기보다, 작은 기능 단위로 자주 커밋하는 게 관리에 유리
4. **.gitignore 사용**
   * 빌드 파일, IDE 설정 파일 등을 제외하고 관리하는 데 유용

**📦 예시 Git 폴더 구조**

text

복사편집

/my-project

/srcㄴ

- main.cpp

- myclass.cpp

- myclass.h

/include

- mylib.h

/tests

- test\_myclass.cpp

/docs

- README.md

.gitignore

CMakeLists.txt

**Git**을 잘 활용하면 리팩토링과 소스 관리가 더 효율적이니까, **기본적인 Git 사용법**도 익히는 게 유리할 거예요. 그래도 궁금한 점 있으면 언제든지 도와드릴 수 있습니다!