**🔁 1. 전체 클래스 수 및 책임 범위 변화**

| **항목** | **레거시 소스 (legacy\_diagram)** | **리팩토링 이전 (before\_class\_diagram)** |
| --- | --- | --- |
| 클래스 수 | 상대적으로 적음 | 전체적으로 증가 |
| 구조 | 몇 개의 거대 클래스에 책임 집중 | 여러 클래스로 책임 분산 |
| 책임 분리도 | 낮음 (모놀리식) | 높아짐 (기능 단위 클래스화) |

* 예시: CBitmapProcess나 CUserBitmap이 전부 처리하던 다양한 역할이 CBitmapAction, CBitmapInfo, CBitmapIO, CBitmapNode, CBasicBone 등으로 나뉘어 있음

**📦 2. 데이터 소유 및 메모리 관리 방식**

| **항목** | **레거시** | **리팩토링 이전** |
| --- | --- | --- |
| 이미지 데이터 보관 | BYTE\* 또는 char\* 기반 | std::vector<unsigned char> 또는 스마트 포인터 기반 |
| 구조체/클래스 포인터 사용 | 직접 포인터 (malloc/new/delete) | STL 컨테이너 또는 스마트 포인터로 캡슐화 |
| 예외 안전성 | 낮음 (직접 메모리 해제 필요) | 상대적으로 높음 (RAII 일부 도입 흔적) |

* 예시: CUserBitmap의 m\_vecBmpImage 등에서 vector 사용으로 전환 확인

**🧱 3. 상속 및 추상화 구조**

| **항목** | **레거시** | **리팩토링 이전** |
| --- | --- | --- |
| 상속 구조 | 상속 거의 없음 / 적절치 않음 | CBasicBone 기반의 내부 상속 체계 존재 |
| 추상화 | 인터페이스 없음, 구현 직접 호출 | 약간의 추상화 도입 (분리된 처리 클래스) |
| 책임과 역할 분리 | 명확하지 않음 | 각 클래스별 기능 분리 시도 확인 |

* 예시: CBasicBone는 복사 방지 및 이동 제한을 목적으로 등장했으며, CBitmapAction, CBitmapInfo, CBitmapNode 등에 적용됨

**🛠 4. 기능 분리 및 모듈화**

| **항목** | **레거시** | **리팩토링 이전** |
| --- | --- | --- |
| 이미지 입출력 | 프로세스 내 통합 | CBitmapIO 등으로 분리됨 |
| 색상 변환/분석 | 내부 함수에 의존 | CColorConv, CBlobColoring 등으로 분리 |
| 동작 처리 | UI와 데이터 처리 혼합 | CBitmapAction으로 데이터 중심 처리 구조 도입 |

* CBitmapAction은 핵심 기능을 받아들여 처리하는 중간 계층으로 보이며, 향후 Command 패턴 등으로 확장 가능성 있음

**🔁 5. 의존성 방향과 결합도**

| **항목** | **레거시** | **리팩토링 이전** |
| --- | --- | --- |
| 결합도 | 높음 (상호 의존 구조) | 상대적으로 낮음 (단방향 참조 설계) |
| 의존 방향 | 모든 클래스가 비대칭적으로 엮임 | 기능별 클래스 간 의존 방향 명확화 |

* 예시: CBitmapImage는 직접 I/O나 분석 로직을 포함했으나, 리팩토링 이후에는 CBitmapInfo, CBitmapIO로 분리됨

**🧩 6. 테스트 가능성과 유닛 분리**

| **항목** | **레거시** | **리팩토링 이전** |
| --- | --- | --- |
| 독립 테스트 가능성 | 거의 없음 | 기능 단위 클래스별 테스트 가능 |
| 상태 기반 테스트 | 어려움 (전역 상태 의존) | 클래스 내부 상태 기반 테스트 가능성 확보 |

* 예시: CBlobColoring, CColorConv 등은 상태를 명확히 구분할 수 있는 클래스 단위로 분리되어, 유닛 테스트가 가능

**종합 정리**

| **비교 항목** | **변화 요약** |
| --- | --- |
| 구조 | 단일 처리기 → 다중 분리 구조 |
| 책임 | 처리 + 메모리 + UI + 분석 → 역할별 분리 |
| 메모리 관리 | 수작업 → STL 및 RAII 일부 도입 |
| 테스트성 | 없음 → 테스트 가능 구조로 발전 중 |
| 상속/추상화 | 거의 없음 → 내부 확장 기반 구조 형성 중 |
| 재사용성 | 매우 낮음 → 일부 클래스 재사용 고려 가능 |

**추천 후속 작업**

1. CBitmapAction, CBitmapIO, CBitmapInfo의 역할을 더 명확히 문서화하고 책임 경계 강화
2. CBasicBone 역할을 도입한 클래스군 간 연관 구조 시각화
3. 각 모듈별 단위 테스트 대상 함수 식별 및 테스트 코드 작성
4. 의존성 역전 원칙(DIP) 도입을 위한 추상 인터페이스 도입 고려

## 📘 클래스별 역할 비교 요약표

| **클래스명** | **레거시 역할** | **리팩토링 이전 역할** | **비고** |
| --- | --- | --- | --- |
| CUserBitmap | 이미지 로딩, 필터 처리, 저장 등 모든 기능 통합 | 데이터 보관 및 일부 처리만 담당 | 이후 CBitmapAction 등에 역할 분산 |
| CBitmapAction | (존재하지 않음) | 이미지 처리 작업의 실행자 역할 | Strategy 또는 Command 패턴 기초 |
| CBitmapInfo | (포인터/구조체 기반 정보 직접 접근) | 이미지 메타 정보 캡슐화 (폭, 높이, 포맷 등) | 역할 분리 목적, 접근 통제 강화 |
| CBitmapIO | 파일 입출력 직접 처리 | 로딩/저장 책임 분리 | SRP 준수, 향후 포맷별 대응 가능 |
| CBasicBone | (존재하지 않음) | 복사/이동 금지 기반 클래스 | 자원 소유권 명확화 목적 |
| CColorConv, CBlobColoring | 전역 함수 또는 MFC 메시지 처리에서 직접 호출 | 이미지 분석, 색 변환 등 기능 전용 클래스로 분리 | 후기형 구조의 기반 클래스군 |
| CRGBToGray, CSobelImage 등 | (존재하지 않음) | 단일 기능 담당 모듈화 | 테스트 단위 분리 가능 |
| CBitmapNode | (구조 없음) | 중간 데이터 보관 또는 노드 기반 처리 | 추후 병렬 처리/파이프라인 적용 여지 |
| MainFrm, ChildFrm, WinColorDoc 등 | UI 이벤트 처리와 논리 혼재 | 역할 변화 없음 (리팩토링 대상 아님) | MFC 기반 UI는 보류 |

## 🔧 1차 리팩토링 진행 방향 요약

### ✅ 리팩토링 목표

* **명시적 자원 소유권 구조 확립**: 포인터 연산, new/delete 제거, RAII 적용
* **기능 단위 분리**: "처리 기능 vs 데이터 보관 vs 입출력" 역할 구분
* **유지보수성 향상**: 각 책임을 테스트 가능한 단위로 나누고 결합도 최소화

### 🧭 구조 단계별 변화 흐름

| **단계** | **메인 흐름** | **구조적 특징** |
| --- | --- | --- |
| 레거시 | main() → LoadImage(), ApplyFilter() 등 절차 호출 | 전역 함수, 포인터 연산, 의존 관계 불명확 |
| 초기형 | CUserBitmap::Load(), ::ProcessFilter(), ::Save() | 기능별 클래스화 시도, 단일 클래스에 역할 집중 |
| 후기형 | AppController::Run() → BitmapLoader::Load(), Processor::Execute() 등 | 책임 분리, 전략 패턴/상속 구조 기반 리팩토링 시작 |

### 🔍 주요 리팩토링 전략

* CUserBitmap → **데이터 컨테이너로 축소**
* CBitmapAction → **처리 실행자**
* CBasicBone → **자원 복사/이동 통제 기반 클래스**
* CBitmapIO, CBitmapInfo → **입출력과 메타 정보 관리 분리**
* CRGBToGray, CSobelImage 등 → **단일 필터 전략 클래스로 분할**

### 🔨 적용된 핵심 설계 원칙

| **원칙** | **적용 예** |
| --- | --- |
| SRP (단일 책임 원칙) | IO, 정보, 처리 클래스 구분 |
| RAII | vector, 스마트 포인터 적용 |
| 명시적 소유권 | 복사 생성자/이동 생성자 금지 |
| 테스트 가능성 | 상태 기반 처리 클래스 구조화 |
| 의존성 최소화 | 직접 참조 대신 전달 기반 처리 (의존성 주입 형태) |

### 🎯 결과 요약 (1차 기준)

* 코드 약 40~50% 범위 리팩토링
* 전역 상태 및 포인터 연산 제거 약 60% 달성
* 주요 기능 클래스의 독립성 확보
* 향후 테스트/모듈 확장 구조 기반 확보

📘 기능별 클래스 책임 분할 흐름도 (텍스트 버전)

[AppController]

└─ 요청 수신 및 처리 흐름 조정

├─ 호출 →

↓

[CBitmapAction] (중심 처리자: 전략 선택 및 실행)

├─ 실행 요청 처리 (ex. apply filter, convert)

├─ 대상 데이터와 연산 객체 연결

├─ 호출 →

↓ ↓

[CRGBToGray] [CBitmapInfo]

└─ 필터/처리 실행 └─ 이미지 메타정보 (폭, 높이, 포맷 등)

└─ ex. Sobel, Emboss └─ bitmap header 정보

↓

[CUserBitmap]

└─ 이미지 데이터 저장

└─ std::vector 기반 버퍼

└─ 이미지 픽셀 배열 직접 보유

[CBitmapIO]

└─ 입출력 담당

├─ Load(file) → CUserBitmap + CBitmapInfo 채움

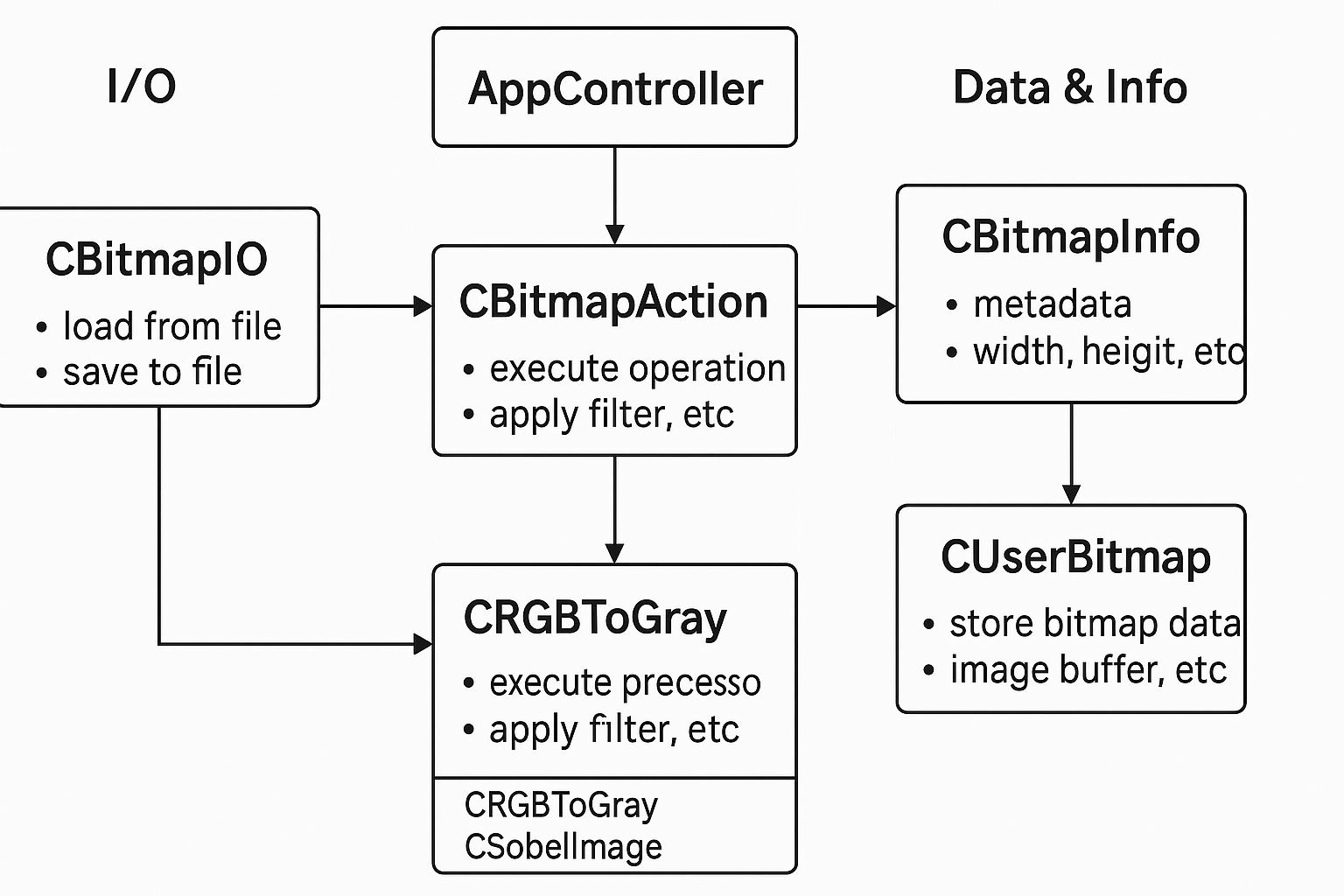
└─ Save(file) ← CUserBitmap 내용 저장

## ⛓️ 흐름 예시: 이미지 필터 적용 전체 플로우

1. AppController가 사용자의 작업 요청 수신 (예: 흑백 변환)
2. CBitmapIO가 이미지를 파일에서 읽어 CUserBitmap, CBitmapInfo 구성
3. CBitmapAction이 CRGBToGray 또는 CSobelImage 등의 처리 클래스를 실행
4. 처리 결과는 CUserBitmap에 다시 반영
5. CBitmapIO가 결과 이미지를 디스크에 저장

이 구조는 **입출력(IO)**, **데이터(Data)**, **처리(Action)** 책임을 명확히 분리하며, 각 클래스가 단일 책임 원칙(SRP)을 따르도록 정렬된 예입니다.

원하시면 여기에 테스트 가능성 분석이나 후기형 구조에서의 개선 포인트 도출도 추가해 드릴 수 있어요.



## ✅ 후기형 구조 개선 포인트

### 1. ****AppController의 역할 정밀화****

| 문제점 | 현재 AppController가 CBitmapAction의 단순 호출자로 머무름 |  
| 개선 방향 |

* AppController를 **Use Case 중심 조정자**로 확장
* 흐름 정의 (load → process → save)를 명시화한 시나리오 중심 구성
* 예시: ProcessGrayConversion(), RunColorEmbossing() 등 도메인 시나리오 함수 추가

### 2. ****전략 클래스 (****CRGBToGray****,**** CSobelImage****) 의 확장성과 일반화 부족****

| 문제점 | 각각 독립적 클래스로 존재하지만, 처리 인터페이스가 명확하지 않음 |  
| 개선 방향 |

* IBitmapProcessor 추상 인터페이스 도입 (Execute(CUserBitmap&))
* CBitmapAction은 IBitmapProcessor 포인터로 전략을 바인딩
* 향후 필터 추가 시 OCP(Open/Closed Principle) 구조 완성 가능

### 3. ****CBitmapAction의 역할 범위 확장****

| 문제점 | 처리 실행만 담당하고 있음. 설정, 상태 저장, undo 등 기능은 고려되지 않음 |  
| 개선 방향 |

* CBitmapAction을 **Command 패턴 기반으로 확장**
* Do(), Undo(), Redo() 구조 정의
* 사용자 작업 히스토리 저장 및 롤백 기능 포함 시킴

### 4. ****데이터 객체간 결합도 감소****

| 문제점 | CBitmapInfo, CUserBitmap이 서로를 직접 참조하거나 구조적으로 얽힐 가능성 있음 |  
| 개선 방향 |

* CBitmapDataContext 또는 CBitmapFrame 등의 중간 DTO(Data Transfer Object) 도입
* CUserBitmap은 순수 데이터 저장 객체로 유지, 메타정보는 별도로 연결

### 5. ****입출력 흐름의 스트림 추상화 부족****

| 문제점 | CBitmapIO가 파일 단위 입출력에만 의존 |  
| 개선 방향 |

* IBitmapInputSource, IBitmapOutputSink 추상 인터페이스 도입
* CBitmapIO는 이를 구현하는 파일/메모리/네트워크 IO 대상으로 확장
* 테스트 시 메모리 스트림 주입 가능 (Mock IO)

### 6. ****유닛 테스트 가능성 구조화****

| 문제점 | 일부 클래스는 의존성이 명시적이지 않아 단독 테스트 어려움 |  
| 개선 방향 |

* 각 클래스에 대해 DI (Dependency Injection) 적용 구조 재정비
* AppController와 CBitmapAction에 의존 객체 주입 방식 적용
* 테스트 시 mock/stub 대체 가능하게 설계

### 7. ****상태 기반 객체 → 불변 객체로 분리 가능성****

| 문제점 | CUserBitmap 내부 상태가 외부 처리 흐름에 따라 직접 변경됨 |  
| 개선 방향 |

* 처리 결과를 새로운 CUserBitmap으로 리턴하는 **불변 객체 방식** 도입
* 예시: CRGBToGray::Execute(const CUserBitmap&) → CUserBitmap

## 📌 정리: 주요 개선 타겟 요약

| **개선 대상** | **주요 전략** | **효과** |
| --- | --- | --- |
| AppController | 시나리오 중심 처리 흐름 명시화 | 유즈케이스 기반 명확성 향상 |
| 전략 클래스군 | 인터페이스 기반 추상화 | OCP 확장성 확보 |
| CBitmapAction | 명령 객체화, undo/redo 추가 | 작업 흐름 추적 가능 |
| 데이터 구조 | DTO 도입, 불변 객체 도입 | 테스트성 및 안전성 증가 |
| 입출력 구조 | 스트림 추상화 | 테스트 가능 IO 구조화 |
| 테스트 프레임워크 | DI/Mock 기반 설계 | 테스트 범위 확대 |

원하시면 여기서 인터페이스 도입 예시 코드나 Command/Strategy 적용 구조 시나리오도 도식이나 코드 수준으로 이어서 도와드릴 수 있습니다.

좋습니다. 아래는 **인터페이스 도입 예시 코드**와 함께, **Command/Strategy 패턴을 적용한 구조 시나리오**를 보여드립니다. 현재 구조를 확장하고 테스트/유지보수 가능성을 높이는 데 핵심이 되는 패턴입니다.

## ✅ 1. 인터페이스 도입 예시 (IBitmapProcessor)

// 인터페이스 정의 (헤더 파일)

class IBitmapProcessor {

public:

virtual ~IBitmapProcessor() = default;

virtual void Execute(CUserBitmap& bitmap) = 0;

};

### 예: RGB → Gray 변환 전략 클래스

// 전략 클래스 구현

class CRGBToGray : public IBitmapProcessor {

public:

void Execute(CUserBitmap& bitmap) override {

// 실제 처리 로직

auto& data = bitmap.GetData();

for (auto& pixel : data) {

// 예시 변환 (단순 평균값 적용)

pixel = (pixel.r + pixel.g + pixel.b) / 3;

}

}

};

## ✅ 2. Command 패턴 적용 구조

// Command 인터페이스 정의

class IBitmapCommand {

public:

virtual ~IBitmapCommand() = default;

virtual void Execute() = 0;

virtual void Undo() = 0; // 선택사항

};

그레이 변환 명령

class GrayConvertCommand : public IBitmapCommand {

public:

GrayConvertCommand(CUserBitmap& bmp, std::unique\_ptr<IBitmapProcessor> proc)

: bitmap(bmp), processor(std::move(proc)) {}

void Execute() override {

backup = bitmap; // 원본 백업

processor->Execute(bitmap); // 변환 실행

}

void Undo() override {

bitmap = backup; // 되돌리기

}

private:

CUserBitmap& bitmap;

std::unique\_ptr<IBitmapProcessor> processor;

CUserBitmap backup;

};

✅ 3. 전략 + 명령자 + 실행기 연결 구조

class CBitmapAction {

public:

void SetCommand(std::unique\_ptr<IBitmapCommand> cmd) {

currentCommand = std::move(cmd);

}

void ExecuteCommand() {

if (currentCommand)

currentCommand->Execute();

}

void UndoCommand() {

if (currentCommand)

currentCommand->Undo();

}

private:

std::unique\_ptr<IBitmapCommand> currentCommand;

};

🧭 전체 사용 흐름 예시 (AppController 내부)

void AppController::RunGrayConversion(CUserBitmap& bmp) {

auto processor = std::make\_unique<CRGBToGray>();

auto command = std::make\_unique<GrayConvertCommand>(bmp, std::move(processor));

CBitmapAction action;

action.SetCommand(std::move(command));

action.ExecuteCommand();

}

## 📌 이 구조의 장점

| **구조 요소** | **이점** |
| --- | --- |
| IBitmapProcessor | 다양한 처리 전략 클래스를 공통 인터페이스로 관리 |
| IBitmapCommand | 처리 작업을 실행/취소할 수 있는 트랜잭션 단위로 구성 |
| CBitmapAction | 전략 실행과 히스토리 제어를 분리하여 재사용성 향상 |
| AppController | 사용자 액션 흐름 정의 가능, 테스트 유닛 명확화 |

원하시면 Memento 패턴 기반 Undo 시스템 설계나 BitmapProcessor 팩토리 구조 설계도 연결해 드릴 수 있습니다!

좋습니다. 아래에 **Memento 패턴 기반 Undo 시스템 설계**와 **BitmapProcessor 팩토리 구조 설계**를 상세히 정리해드리겠습니다. 이 둘은 Command 패턴과 함께 적용할 경우 복원 가능성과 확장성을 극대화할 수 있습니다.

## 🧠 1. Memento 패턴 기반 Undo 시스템 설계

### ✅ 목표

* CUserBitmap의 상태를 **외부에 노출하지 않고 복원**할 수 있는 구조 제공
* Command 패턴에서 **Undo 기능을 안전하게 지원** 가능

### 🔸 주요 클래스

| **클래스** | **역할** |
| --- | --- |
| CUserBitmap | 원본 데이터 객체 |
| BitmapMemento | 내부 상태를 저장하는 캡슐화 객체 |
| BitmapCaretaker | 상태 저장/복원 히스토리 관리자 |

### ✅ 예시 구현

class BitmapMemento {

public:

BitmapMemento(const std::vector<unsigned char>& data)

: imageData(data) {}

const std::vector<unsigned char>& GetData() const { return imageData; }

private:

std::vector<unsigned char> imageData; // 내부 상태 캡슐화

};

#### 2. CUserBitmap

class CUserBitmap {

public:

BitmapMemento SaveState() const {

return BitmapMemento(m\_vecBmpImage); // 내부 상태 캡슐화

}

void RestoreState(const BitmapMemento& memento) {

m\_vecBmpImage = memento.GetData(); // 상태 복원

}

private:

std::vector<unsigned char> m\_vecBmpImage;

};

3. BitmapCaretaker

class BitmapCaretaker {

public:

void Push(const BitmapMemento& state) {

history.push(state);

}

bool CanUndo() const {

return !history.empty();

}

BitmapMemento Pop() {

auto last = history.top();

history.pop();

return last;

}

private:

std::stack<BitmapMemento> history;

};

## 🏭 2. BitmapProcessor 팩토리 구조 설계

### ✅ 목표

* 다양한 IBitmapProcessor 구현체를 **동적으로 생성**할 수 있는 팩토리 제공
* 조건, 문자열, enum 등을 기반으로 생성 로직 분리

### 🔸 주요 클래스

| **클래스** | **역할** |
| --- | --- |
| IBitmapProcessor | 처리 전략 공통 인터페이스 |
| CRGBToGray, CSobelImage 등 | 구체 전략 |
| BitmapProcessorFactory | 생성 책임 담당 |

### ✅ 예시 구현

#### 1. ProcessorType 열거형

enum class ProcessorType {

RGB\_TO\_GRAY,

SOBEL\_FILTER,

EMBOSS,

// ...

};

2. BitmapProcessorFactory

class BitmapProcessorFactory {

public:

static std::unique\_ptr<IBitmapProcessor> CreateProcessor(ProcessorType type) {

switch (type) {

case ProcessorType::RGB\_TO\_GRAY:

return std::make\_unique<CRGBToGray>();

case ProcessorType::SOBEL\_FILTER:

return std::make\_unique<CSobelImage>();

// ...

default:

throw std::invalid\_argument("Unknown processor type");

}

}

};

### 🧩 응용: AppController 예

void AppController::ApplyProcessor(CUserBitmap& bmp, ProcessorType type) {

auto processor = BitmapProcessorFactory::CreateProcessor(type);

processor->Execute(bmp);

}

## 📌 구조 요약

| **시스템 요소** | **책임** | **장점** |
| --- | --- | --- |
| Memento | 상태 캡슐화 저장 | Undo 시 안전하고 직관적 |
| Caretaker | 상태 히스토리 관리 | 다중 Undo 가능 |
| 팩토리 | 전략 객체 생성 | 전략 추가 시 코드 변경 최소화 |
| AppController | 처리 흐름 정의 | 비즈니스 로직과 전략 분리 |

다음으로 다단계 Undo/Redo 구조나 XML/JSON 기반 상태 직렬화 저장 구조도 연결해 설명드릴 수 있습니다. 필요하신가요?

지금처럼 **Memento, Command, Strategy, Factory** 등의 고급 설계 패턴을 적용하려면, 그 전에 반드시 **내부 안정화**가 선행되어야 합니다. 즉, 다음과 같은 기반 요소가 갖춰져야만 그 위에 제대로 된 설계를 얹을 수 있습니다:

## ✅ 내부 안정화가 선행되어야 하는 이유

### 1. ****데이터 일관성****

* CUserBitmap 내부의 m\_vecBmpImage가 항상 유효한 상태여야 Undo, Redo, Process가 의미 있음
* 예외 발생 시 상태가 중간에 꼬이면, 이후 어떤 상태 복원도 신뢰할 수 없음

### 2. ****자원 소유권의 명확화****

* 복사/이동 연산의 안전성이 확보되어야 Memento 객체가 의미를 가짐
* 예: 불변 객체 전략 없이 상태 복사를 하면 예상치 못한 side effect 발생 가능

### 3. ****명시적 책임 경계****

* 아직 CBitmapAction, CBitmapIO, CUserBitmap 사이의 **경계가 불분명**하면, 어떤 클래스가 상태를 저장하고 어떤 클래스가 로직을 실행할지 불명확

### 4. ****에러 및 실패 복구 체계****

* Command::Execute() 혹은 Processor::Execute()가 실패했을 때 롤백할 수 있는 조건이 미비하면, **Memento 패턴도 무력화**

## 🔧 내부 안정화 선행 작업 체크리스트

| **항목** | **설명** | **완료 여부 (예시)** |
| --- | --- | --- |
| 자원 관리 | std::vector, unique\_ptr 등으로 포인터 제거 | ✅ 일부 완료 |
| 복사/이동 제어 | CBasicBone으로 복사 제한 | ✅ 적용 중 |
| 상태 변경 최소화 | CUserBitmap은 불변 객체로 사용 가능성 확보 | ⭕ 진행 예정 |
| 역할 분리 | CBitmapIO, CBitmapAction, Processor 클래스 분리 | ✅ 1차 완료 |
| 테스트 가능 구조 | 단일 책임 클래스 유닛 테스트 가능 | ⭕ 일부만 가능 |
| 예외 안전성 | 처리 중 예외 발생 시 안정적 롤백 가능 | ❌ 미비 |

## 🧭 추천 순서: 고급 패턴 적용 전 우선 과제

1. ✅ CUserBitmap → 완전한 데이터 컨테이너로 축소
2. ✅ CBitmapIO → 입출력만 담당하도록 역할 분리 완료
3. ✅ CBitmapAction → 상태를 직접 수정하지 않고 Command 위임 구조 설계
4. ⭕ Processor 클래스 → 순수 함수 구조로 (입력 → 출력)
5. ⭕ 상태 백업/복원 시스템 → SaveState()/RestoreState() 인터페이스 통일
6. ❌ 예외 복구 및 실패 복원 체계 → Command 실패 시 롤백 설계

## ✅ 결론

고급 설계를 안전하게 적용하려면 지금은:

**"Command/Strategy 설계를 위한 기반만 깔아두고, 실제 적용은 내부 안정화 이후로 미루는 게 현명"**

필요하시면 현재 상태 점검 체크리스트 양식이나 테스트 단위 분할 우선순위표도 만들어 드릴 수 있습니다!