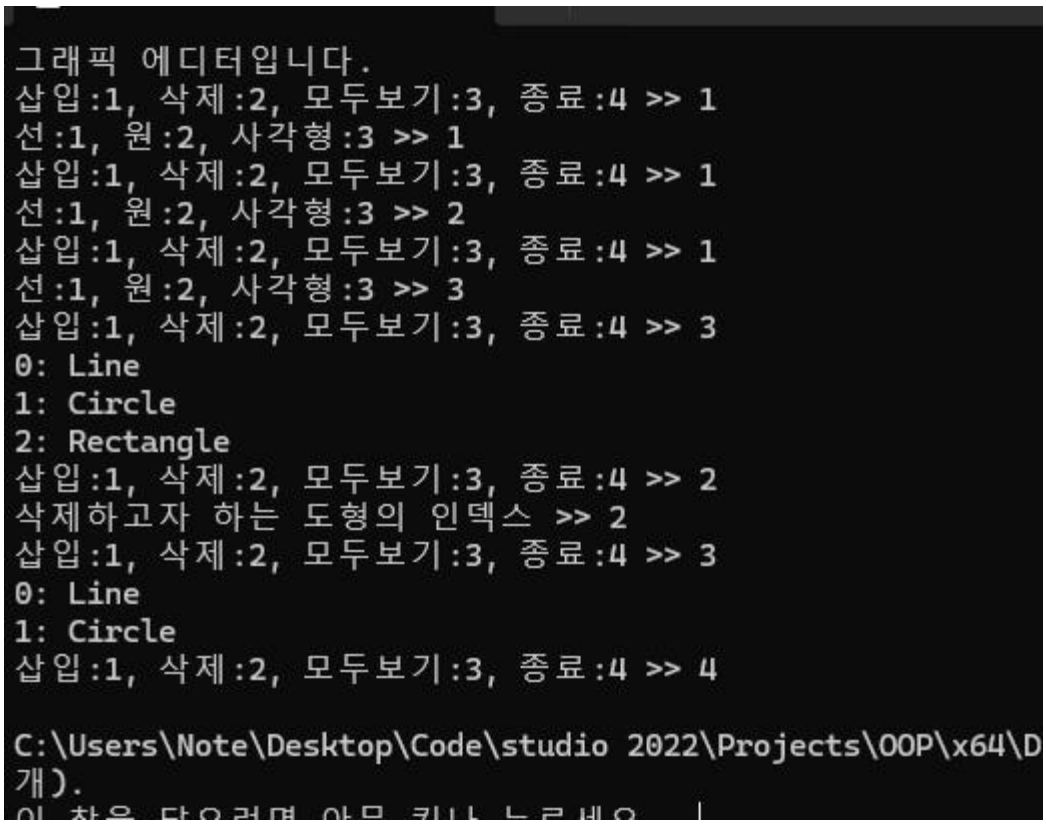
객체지향프로그래밍[02]

202304035 김유진

202104354 지현준

#실행 결과



#문제 정의

GraphicEditor 클래스에 도형(Shape)을 삽입, 삭제, 모두보기, 프로그램을 종료하는 기능을 구현한다. 문제는 GraphicEditor 클래스가 사용자 입력에 따라 도형을 삽입하거나 삭제하고, 삽입된 모든 도형의 정보를 출력하는 기능을 동적 객체 관리 및 다형성을 활용하여 만들어야 한다.

Shape 클래스는 순수 가상 함수인 draw()를 가지는 추상 클래스로 정의되며, 이를 상속받는 Circle(원), Line(선), Rect(사각형) 클래스가 각 도형의 특성에 맞는 draw()를 구현한다.

GraphicEditor 클래스는 동적 할당할 도형 객체를 저장하기 위해 동적 객체 관리가 가능한 vector<Shape*>를 사용하며, 이 벡터를 통해 다양한 도형 객체들을 보다 더 쉽게 동적으로 생성, 삭제, 출력할 수 있다. 프로그램은 사용자 입력에 따라 동작하며, 삽입할 도형의 종류나 삭제할 도형의 인덱스는 사용자가 입력한 것에 따라 선택된다.

#문제 해결 방법

1. Shape 클래스는 추상 클래스로 설계되었으며, 순수 가상 함수인 draw()를 선언하여 모든 파생 클래스가 이를 반드시 구현하도록 강제했다. 이를 통해 도형의 공통적인 인터페이스를 정의했으며, Circle, Line, Rect 클래스는 각각의 도형을 표현하며 draw()를 구현하여 자신만의 고유한 동작을 정의했다.

2. Shape 클래스는 추상 클래스이므로 객체를 직접 생성할 수 없다. 대신, 이를 상속받은 각 파생 클래스에서 draw() 메소드를 구현해야만 인스턴스를 생성할 수 있다. Shape 클래스의 paint() 메소드는 draw()를 호출하며, 실행 시점에 각 도형 클래스의 draw() 메소드가 동적으로 호출되도록 설계되었다. 이를 통해 다형성을 활용하여 Shape* 타입으로 다양한 도형 객체를 일관되게 관리할 수 있다.

3. GraphicEditor 클래스의 멤버 객체로 vector<Shape*> v를 선언하여 생성된 도형 객체들은 이 vector 객체 v를 통해 동적으로 관리되도록 한다. 이때, Shape*로 지정하는 것은 도형들을 동적으로 할당하여 생성할 것이기 때문이다. 도형을 삽입할 때는 사용자의 선택에 따라 해당 도형 클래스의 객체를 동적으로 생성하고, 이를 삽입되는 값을 벡터의 맨 마지막에 삽입하는 push_back() 멤버함수를 사용해 vector에 추가한다. 삭제 기능에서는 사용자가 삭제할 도형의 인덱스를 입력하면, 유효성 검사를 거친다. 그 후, 해당 인덱스의 도형을 동적 해제하고 벡터에서 삭제한다.
4. 사용자 입력 처리는 UI 클래스의 정적 메소드를 통해 이루어진다. 삽입, 삭제, 조회와 관련된 입력을 처리하는 메소드는 정적(static)으로 구현되어 UI 클래스의 인스턴스를 생성하지 않고도 호출할 수 있다. 이를 통해 코드의 모듈화와 재사용성을 높였다.
5. GraphicEditor 클래스는 프로그램의 전반적인 흐름을 제어한다. call() 메소드를 통해 사용자 입력을 반복적으로 처리하며, 입력된 명령에 따라 삽입, 삭제, 조회 또는 종료 기능을 수행한다. 프로그램 종료 시에는 GraphicEditor 클래스의 소멸자를 통해 vector에 남아 있는 모든 도형 객체를 삭제하고 메모리를 해제함으로써 안정성을 보장했다.
6. 모두 보기 기능의 경우 for문을 벡터의 크기인 v.size()만큼 돌리도록 하여 벡터 v에 들어있는 모든 도형들을 인덱스와 함께 출력하도록 했다. 이때, 벡터는 배열처럼 사용할 수 있으며 벡터 v에는 도형들의 포인터가 담겨있기 때문에 화살표 연산자로 paint() 함수에 저근하여 자신의 도형을 그리도록 한다.
7. GraphicEditor의 소멸자는 for문을 벡터의 크기인 v.size()만큼 돌리도록 하여 동적 할당한 모든 도형들을 메모리 해제하도록 했다.

#아이디어 평가

1. Shape 클래스를 추상 클래스로 설계하고, 다형성을 적극적으로 활용한 점은 매우 효율적이었다. 이를 통해 도형 추가 시 기존 코드를 수정하지 않고 새로운 클래스를 쉽게 확장할 수 있다. 또, vector를 사용한 도형 관리 방식은 삽입, 삭제, 조회를 링크드 리스트를 이용했을 때보다 보다 편리하게 일관되게 처리할 수 있었다. 뿐만 아니라 벡터를 통해 코드를 더 간략하면서도 직관적으로 작성할 수 있었다.
2. UI 클래스의 정적 메소드를 활용한 사용자 입력 처리 방식은 프로그램의 흐름을 명확히 하면서도 코드의 재사용성을 높이는 데 효과적이었다. GraphicEditor 클래스에서 도형 관련 동작을 모듈화한 점도 프로그램의 유지보수성을 높이는 데 기여했다.
3. GraphicEditor의 소멸자를 정의함으로써 프로그램 종료 시 벡터 크기만큼 for문을 돌렸다. 이 과정에서 벡터의 모든 원소들을 순회하며 동적 할당한 도형들을 메모리 해제하도록 한 알고리즘을 통해 메모리 누수를 방지할 수 있었다.
4. 컨테이너 원소에 대한 포인터인 iterator를 sh로 선언하여 벡터의 크기를 동적으로 관리할 수 있었던 부분과 벡터를 배열처럼 사용해 동적 할당한 도형을 메모리 해제한 부분에 대해 삭제 기능을 효율적으로 수행할 수 있었다.

#문제를 해결한 키 아이디어 또는 알고리즘 설명

이번 문제의 핵심은 벡터를 이용한 삭제 기능을 구현한 알고리즘이라고 생각한다. 벡터의 장점을 그대로 보여주는 부분이기 때문이다. 특정 인덱스에 대한 도형을 벡터에서 삭제해야 하기 때문에 컨테이너 원소에 대한 포인터인 iterator를 sh로 선언한다. (erase() 함수는 iteratro 포인터 변수를 통해 호출할 수 있기 때문) sh가 begin() 함수를 통해 벡터의 첫 번째 원소에 대한 참조값에 삭제할 인덱스 값을 더한다. 그 결과, sh는 삭제할 원소를 가리키게 된다. 벡터에서 해당 인덱스의 원소를 삭제하기 전, 벡터는 배열처럼 사용될 수 때문에 delete v[num]으로 해당 도형의 동적 할당을 해제한다. 또한 erase() 함수를 통해 sh가 가리키는 원소, 즉 삭제할 인덱스의 도형을 벡터에서 삭제하고 자동으로 벡터를 조절하도록 한다. 이를 통해 메모리 누수를 방지하며 동적 할당된 객체를 동적으로 안전하게 관리한다.