문제해결 및 실습 : Java

과제 3 보고서

세종대학교

제출일	2024.11.30	전공	소프트웨어학과
과목	문제해결및실습:Java	학번	21011746
담당교수	박상일 교수님	이름	양현석

1. 계산기 프로그램

a) 문제 분석

Java Swing을 이용하여 주어진 계산기 프로그램과 가능한 흡사하게 만드는 과제입니다. 가장 먼저 연산을 정확하게 해야 하고, 디자인적인 부분에 대해서도 신경 써야 합니다. 디자인적인 부분 첫 번째, 창 크기가 바뀜에 따라 비율을 유지하며 계산기의 사이즈가 변동되야 합니다. 두 번째, 폰트의 색상을 변경하고 그림자 효과를 구현해야 합니다. 세 번째, 일부 패널들에 그라데이션 효과를 구현해야 합니다.

b) 문제 풀이

가장 먼저 계산기 기능을 만들어 내기 위해 Stack을 사용해야겠다는 생각을 했습니다. 기본적으로 '+', '-' 입력이 들어왔을 때 입력된 숫자가 있다면 push하고 '='이 입력으로 들어오면 pop해서 연산하고 push 하는 방법입니다. 물론 상황에 따라 수정할 부분이 있기 때문에 이에 대한 코드는 이후에 버튼에서 설명하도록 하겠습니다.

이번에는 필요한 부분을 나누어보겠습니다. 이 문제를 풀기 전에 가능한 주어진 계산기의 형태와 흡사하게 구성하고, 이전에 웹 프로그래밍과 유니티의 UI를 구성해본 경험을 바탕으로 배치를 돕는 Wrapper, 즉 기능은 없지만 배치를 돕는 역할을 하는 객체를 이용하는 것으로 구성해보았습니다. 먼저 상단인 디스플레이 부분과 하단인 버튼 부분으로 크게 나누었습니다. 이 상단과 하단의 비율은 3 : 7로 나누었습니다.

상단 부분부터 살펴보자면, 가장 바깥쪽부터 upperWrapper, displayBackgroundPanel, dispalyPanel로 panel이 나누어지고, 글씨가 보여지는 displayLabel이 존재합니다. Panel부터 살펴보면 JPanel를 상속받는 Hw3Panel 클래스를 제작하였습니다. 이 panel 클래스의 핵심 기능은 그라데이션 부분을 추가하는 것이었습니다. 주어진 예제를 보면 그라데이션 종류가 여러 가지로, 이를 구현하기 위해서 paintComponent를 overriding하고 생성자를 통해 입력 받는 mode로 그려지는 그라데이션 효과를 구분했습니다. 다음은 코드입니다.

```
//  0 : 아무것도 없음, 1 : 회색으로 위로, 2 : 검정으로 위로, 3: 하얗게 위로
(그라데이션 효과)
int mode;

Hw3Panel(Color color, int mode) {
    this.color = color;
    this.mode = mode;
}

@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);

    Graphics2D g2 = (Graphics2D) g;
    g2.setColor(color);
```

```
    GradientPaint gp;

    //  mode 값에 따라 각기 다른 그라데이션 효과를 부여
    if (mode == 1) {
        gp = new GradientPaint(0, getHeight(), color, 0, 0,
Color.LIGHT_GRAY);
        g2.setPaint(gp);
    }
    else if (mode == 2) {
        gp = new GradientPaint(0, getHeight(), color, 0, 0, Color.GRAY);
        g2.setPaint(gp);
    }
    else if(mode == 3){
        gp = new GradientPaint(0, getHeight(), color, 0, 0, Color.WHITE);
        g2.setPaint(gp);
    }

    g2.fillRect(0, 0, getWidth(), getHeight());
}
```

맨 위의 주석에서 알 수 있듯이 0, 1, 2, 3 총 4가지의 모드가 존재하고 각각 다른 그라데이션 효과를 그려낼 수 있습니다. 또한 패널의 색상을 예제와 흡사하게 구성하기 위해서 맥북의 기본 프로그램인 디지털 컬러 측정기를 이용해서 RGB 값을 동일하게 생성자에 넣었습니다.

상단 부분의 레이어에 대해 살펴 보겠습니다. upperWrapper는 GridBagLayout을 사용하였고, 다른 나머지 panel들은 BorderLayout을 사용했습니다. upperWrapper에서 GridBagLayout을 사용한 이유는 확장 비율을 적용할 수 있기 때문이었습니다. Weightx, weighty 변수를 조절하고 fill을 GridBagConstraints.BOTH로 설정하게 되면 upperWrapper가 가지고 있는 패널이 upperWrapper 크기에 따라 자동적으로 변환할 수 있습니다. 나머지 panel들은 padding을 부여하기 위해서 borderLayout을 적용했고, 이들의 값은 초기에 임의로 설정했습니다.

이번에는 글자가 보여지는 displayLabel입니다. 연산을 수행하기 위해 선언한 String 클래스 buff라는 객체를 그대로 담게되는 displayLable은 항상 오른쪽을 기준으로 채워지고 이를 품고 있는 displayPanel의 높이의 중간에 위치하기 때문에 이 label의 Alignment를 조절해주었습니다. 오른쪽으로 정렬하기 위해서는 SwingConstants.Right라는 값을 이용해야 했습니다. 다음은 upperWrapper와 그 내부의 panel, label을 설정하는 메소드입니다.

```
//  디스플레이 영역인 upperWrapper 설정
private void setUpperWrapper() {
    upperWrapper = new Hw3Panel(Color.GRAY, 3);
    upperWrapper.setLayout(new GridBagLayout());    //  Wrapper의 Layout
설정

    //  디스플레이 Background Panel
    displayBackgroundPanel = new Hw3Panel(new Color(101, 101, 101), 1);
    displayBackgroundPanel.setLayout(new BorderLayout());

    //  디스플레이 Panel
    displayPanel = new Hw3Panel(new Color(151, 159, 150), 2);
    displayPanel.setLayout(new BorderLayout());
```

```
    //  디스플레이 Label
    displayLabel = new Hw3Label("0");
    displayLabel.setHorizontalAlignment(SwingConstants.RIGHT);
    displayLabel.setVerticalAlignment(SwingConstants.CENTER);

    //  각 Panel 에 Padding 추가
    upperWrapper.setBorder(new EmptyBorder(5, 5, 5, 5));
    displayPanel.setBorder(new EmptyBorder(10, 10, 10, 10)); // 디스플레이
내부 패딩
    displayBackgroundPanel.setBorder(new EmptyBorder(15, 15, 15, 15)); //
배경 패딩

    displayPanel.add(displayLabel, BorderLayout.CENTER);
    displayBackgroundPanel.add(displayPanel, BorderLayout.CENTER);

    // GridBagConstraints 설정
    GridBagConstraints gbc = new GridBagConstraints();
    //  시작 위치
    gbc.gridx = 0;
    gbc.gridy = 0;
    //  확장 비율
    gbc.weightx = 1.0;
    gbc.weighty = 1.0;
    //  크기 변경 적용
    gbc.fill = GridBagConstraints.BOTH;

    upperWrapper.add(displayBackgroundPanel, gbc);
}
```

이 코드를 작성할 때는 GridBagLayout과 label의 Alignment를 수정하는 데에 필요한 사용법을 구글 검색과 GPT에서 참고하였습니다.

다음은 하단 부분입니다. 하단 부분에서는 숫자 키패드와 연산자 버튼이 존재하는 구역입니다. 먼저 버튼에 들어가는 String 배열을 구성하고, 0 ~ 9 그리고 각 연산자의 역할을 다르게 하기 위해 각 버튼의 addActionListener을 람다식으로 구성했습니다. 또한 이 버튼을 담아주는 lowerWrapper라는 panel을 추가로 사용했습니다. 이 lowerWrapper는 버튼의 배열과 같은 Layout을 가장 쉽게 구현하게 도와주는 GridLayout을 사용했습니다.

이제 String 배열의 원소들을 읽어오면서 각 버튼을 구현합니다. 버튼 리스트는 저장되며, 각 숫자 키패드는 현재 연산을 저장하는 String buff에 입력을 돕습니다. 단, buff의 입력이 0, 사실상 없을 경우에는 0을 지우며 저장하고 아닐 경우에는 뒷자리에 숫자를 계속해서 붙이게 됩니다. 이때 람다식으로 구성하였기에, effectively final에 대해서도 신경 써야 했습니다. 다음은 그 코드입니다.

```
//  숫자 키
case "0", "1", "2", "3", "4", "5", "6", "7", "8", "9":

    //  익명 클래스에서의 effectively final 을 위한 변수
    int finalNum = i;

    //  lambda expression 사용
    buttonList[i].addActionListener((e) -> {
```

```
        //  buff가 "0"일 때 "0"을 없애기 위한 코드
        if (buff.equals("0")) {
            buff = buttons[finalNum];
        }
        else {
            buff += buttons[finalNum];
        }

        displayLabel.setText(buff);
    });
    break;
```

이번에는 각 연산자에 대해서 작성해보겠습니다. 먼저 연산자는 C, +, -, = 순서로 구분하고, C는 간단하게 구현이 가능했습니다. Stack에 존재하는 모든 원소를 제거하고, buff를 초기화하며, 현재 입력된 연산자도 초기화 시켰습니다. 이때 현재 입력된 연산자는 작성된 코드에서 현재 혹은 마지막으로 입력된 연산자인 curOperator로 사용되어 연산자를 여러 번 입력하는 등의 상황에 유연하게 대처하도록 했습니다. (+ -> "+", - -> "-", = -> "" 으로 저장)

+, - 연산은 사실상 동일한 로직을 사용합니다. 스택이 비어있을 경우에는 현재 입력된 값을 첫 번째 피연산자로 구분하여 스택에 push 후 buff를 초기화 합니다. 그러나 스택이 비어있지 않을 경우가 존재합니다. 이때는 두 가지 경우로 나누어 확인합니다. 먼저 buff가 0 즉, 비어있다고 간주될 경우에는 버튼 입력을 무시합니다. 아니라면, 현재 buff의 있는 값은 두 번째 피연산자라 상정하고 스택에 push 후 두 피연산자를 pop하여 연산 후 buff에 저장하고 다시 스택에 push 합니다. 이때 주의할 점은 다음 값의 입력을 받기 위해 displayLabel의 text를 buff로 바꿔준 후 buff는 초기화 시켜야 합니다.

= 연산은 조금 더 신경 써야 하는 부분이 존재했습니다. 첫 번째로 스택이 비어있거나 buff가 비어있다고 간주될 경우 입력을 무시합니다. 두 번째로 마지막 연산자로도 사용되는 curOperator가 ""로 비어있거나 "="이 마지막 연산으로 구분되는 경우에도 마찬가지로 입력을 무시합니다. 만일 두 경우가 아니라면 마지막 연산자를 기준으로 각기 다른 연산을 스택에 있는 첫 번째 피연산자와 buff에 존재하는 두 번째 피연산자를 연산하게 됩니다.

하나 더 신경써야 했던 부분은 입력이 없는 버튼이 하단에 2개 존재했다는 것입니다. 이는 버튼 자체의 기능을 Disable시켜서 버튼 배열에는 존재하지만 실제로는 사용할 수 없도록 하였습니다. 이러한 방식으로 정상적으로 작동하는 계산기를 구현할 수 있습니다.

예제를 보면 버튼을 감싸고 있는 Wrapper가 더 존재하는 것을 확인할 수 있는데 이는 JPanel을 익명 클래스로 제작하고 paintComponent를 Overriding하고, 버튼을 이 패널에 넣고 이 패널을 lowerWrapper에 넣는 방식으로 구현했습니다. 여기서는 따로 Grapics2D를 사용하지 않았습니다.

```
//  버튼을 감싸고 있는 Wrapper 생성
JPanel buttonWrapper = new JPanel() {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.WHITE);
        g.fillRect(0, 0, getWidth(), getHeight());
```

```
    }
};
```

폰트에 대한 설명입니다. 예제에서는 폰트에 그림자 효과를 추가해놓은 것을 확인할 수 있었습니다. 이 그림자를 넣는 부분은 이 과제에서 가장 구현하는데 시간이 오래걸렸던 부분인 것 같습니다. 또한 사용법을 모르거나, 제대로 작동하지 않는 부분이 많아서 적극적으로 구글 검색을 통해 구현하려고 노력했습니다. 폰트에 대한 디자인이 들어가는 부분은 displayLabel과 버튼의 text입니다. 이 label과 버튼은 각각 Hw3Label, Hw3Button으로 JLabel과 JPanel을 상속받아서 구현했습니다.

paintComponent를 Overriding하여 구현하는 것이 핵심인데 이때, 폰트의 사이즈를 확인하기 위해 FontMetrics라는 클래스를 사용했습니다. 이는 텍스트의 전체적인 길이, 상하좌우 여백을 알 수 있는 getInsets, 기본 높이에서 상승된 만큼의 폭을 알려주는 getAscent, 하강된 만큼의 폭을 알려주는 getDescent 등 다양한 강력한 메소드들을 제공합니다. 이와 그림자를 배치할 변위 shadowOffset 변수를 사용해서 버튼 같은 경우 글씨가 가운데에 오도록 배치했습니다. 또한 폰트의 색상을 바꾸기 위해서 생성자에서 Color 값을 받고 폰트 색상을 저장하는 setForeground를 사용했습니다. 하지만 Label은 구현이 조금 달랐습니다. 한 가운데에 오는 버튼과 달리 오른쪽부터 차곡차곡 쌓여지는 텍스트의 형태이기 때문에 배치도 다르게 하는 것 뿐만 아니라, 만일 super.paintComponent()를 호출하게 되면 일정 글자 수가 넘어가면 글자가 깨져버리는 문제도 발생했기에 super의 method를 부르지 않고도 정상적으로 작동하도록 구현하려 했습니다. 다음은 그 코드입니다.

```
@Override
    protected void paintComponent(Graphics g) {
//        super.paintComponent(g); 은 생략한다.
        //  Label 의 글씨와 위치 보정을 수동으로 하고 있고, 위의 method 를 호출할 시,
일정 글자 수가 넘어가면 문제가 발생

        Graphics2D g2 = (Graphics2D) g;
        //  안티 엘리에이싱 효과
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);

        //  Font 의 크기를 정밀하게 알기 위한 FontMetrics -> 그림자 배치용
        FontMetrics metrics = g2.getFontMetrics(getFont());

        //  현재 텍스트를 가져오기
        String text = getText();
        //  전체 string 의 폭을 가져오기
        int textWidth = metrics.stringWidth(text);

        //  getInsets() : 상하좌우 여백을 가져올 수 있음
        //  getAscent() : 기본 폭에서 상승된 만큼의 폭을 가져옴
        //  getDescent() : 기본 폭에서 하강된 만큼의 폭을 가져옴
        int x = getWidth() - textWidth - getInsets().right; // 오른쪽 정렬
        int y = (getHeight() + metrics.getAscent() - metrics.getDescent())
/ 2;
```

```
        // 그림자 텍스트 그리기
        g2.setColor(Color.WHITE);
        g2.drawString(text, x + shadowOffsetX, y + shadowOffsetY);

        // 본문 텍스트 그리기
        g2.setColor(getForeground());
        g2.drawString(text, x, y);
    }
```

수업 시간에 배운 안티엘리에이싱 효과도 구현했고, super의 메소드를 호출하지 않고도 정상적으로 작동함을 확인했습니다.

마지막으로 가장 중요한 원소들의 재배치입니다. 총 3가지 부분으로 나뉘는데 먼저 클래스 내에 사용된 ASPECT_RATIO를 제외한 static final 변수들은 모두 디버깅을 위한 변수이며 실제로는 사용되지 않는다는 점을 알려드립니다.

비율을 유지하며 원소들을 재배치하는 것은 calculatorPanel의 크기를 조정하며 만들었습니다. 하지만 이 calculatorPanel의 사이즈가 외부 Layout의 영향으로 무시될 수 있었기에 preferredSize를 설정하는 방식으로 코드를 구성했습니다. 먼저 비율을 고려하여 높이와 너비 중 더 큰 값을 가질 수 있는 값을 기준으로 나머지 값을 조절합니다. 이후 adjustComponent 메소드 안에서 모든 원소들의 크기, 패딩 등을 전체 화면에 대한 비율로 조정했습니다. 따로 변수를 만들지 않고 화면에 대한 비율로 설정한 이유는 그것이 가장 효율적으로 크기를 정할 수 있다고 판단했기 때문입니다. 이후에 내부의 모든 원소를 revalidate, repaint하게 됩니다. 이는 JFrame를 상속받는 JavaHw3에 addComponentListener를 overriding하여 maintainAspectRatio를 호출하도록 구성하였습니다. 다음은 maintainAspectRatio와 adjustComponents의 코드입니다.

```
// 창 크기가 변해도 3 : 4 을 유지하도록 함
private void maintainAspectRatio() {
    // Content pane 의 크기 가져오기 (Content pane 을 사용한 이유는 보고서에 기술)
    int frameWidth = getContentPane().getWidth();
    int frameHeight = getContentPane().getHeight();

    // 계산기 패널의 최대 크기 계산 (3:4 비율 유지) -> width, height 을 비교하여
가장 큰 값을 기준으로 설정
    int newWidth = frameWidth;
    int newHeight = (int) (newWidth / ASPECT_RATIO);

    if (newHeight > frameHeight) {
        newHeight = frameHeight;
        newWidth = (int) (newHeight * ASPECT_RATIO);
    }

    // 계산기 패널의 크기 설정 (실제 크기 설정을 위한 Preferred size 설정)
    calculatorPanel.setPreferredSize(new Dimension(newWidth, newHeight));

    // 내부 요소들에 대한 재배치
    adjustComponents(newWidth, newHeight);

    // 전체 레이아웃 다시 적용
    getContentPane().revalidate();
```

```
    getContentPane().repaint();
}
//  폰트 크기 및 비율 자동 변경
private void adjustComponents(int panelWidth, int panelHeight) {
    //  폰트 크기 비율 계산
    int newDisplayFontSize = panelHeight / 7;
    displayLabel.setFont(new Font("Arial", Font.PLAIN,
newDisplayFontSize));

    int newButtonFontSize = panelHeight / 12;
    for (var button : buttonList) {
        button.setFont(new Font("Arial", Font.BOLD, newButtonFontSize));
    }

    //  패딩 조정 (디스플레이 패널) -> 패딩을 조절하면 알아서 창이 바뀐다.
    int newDisplayPanelPadding = panelWidth / 40;
    //  최소 크기 제한 (화면을 너무 줄일 시 없어질 수 있음)
    if (newDisplayPanelPadding < 5) newDisplayPanelPadding = 5;
    displayPanel.setBorder(new EmptyBorder(newDisplayPanelPadding,
newDisplayPanelPadding, newDisplayPanelPadding, newDisplayPanelPadding));

    //  패딩 조정 (디스플레이 Background Panel)
    int newDisplayBackgroundPadding = panelWidth / 30;
    //  최소 크기 제한
    if (newDisplayBackgroundPadding < 5) newDisplayBackgroundPadding = 5;
    displayBackgroundPanel.setBorder(new
EmptyBorder(newDisplayBackgroundPadding, newDisplayBackgroundPadding,
newDisplayBackgroundPadding, newDisplayBackgroundPadding));

    //  upperWrapper 패딩 조절 (내부 패널의 패딩을 기준으로 조정)
    upperWrapper.setBorder(new EmptyBorder(newDisplayBackgroundPadding /
3, newDisplayBackgroundPadding / 3, newDisplayBackgroundPadding / 3,
newDisplayBackgroundPadding / 3));

    //  lowerWrapper 패딩 조절 (내부 패널의 패널을 기준으로 조정)
    lowerWrapper.setBorder(new EmptyBorder(newDisplayBackgroundPadding /
15, newDisplayBackgroundPadding / 15, newDisplayBackgroundPadding / 15,
newDisplayBackgroundPadding / 15));

    //  재배치
    calculatorPanel.revalidate();
    calculatorPanel.repaint();
}
```

이제 upper, lowerPanel의 배치입니다. Frame의 layout은 GridBagLayout으로 구성했고, 이는 upper와 lower의 비율을 3 : 7로 구성하기 위함입니다. Upper에는 weighty를 0.3으로 lower에는 0.7로 주고 frame에 추가하게 되면 비율을 유지하도록 layout을 설정할 수 있었습니다.

c) 시행 착오

이 프로그램을 제작하면서 정말 많은 시행 착오와 어려움이 있었습니다. MFC나 여타 다른 프레

임 워크를 통해 만들어 보았던 것들과 달리 배치의 원리가 달라 소요되는 시간 또한 오래걸렸습니다. 그 중 시간을 가장 많이 썼던 시행 착오들에 대해서만 알아보겠습니다.

첫 번째는 폰트입니다. 폰트의 그림자를 구현하기 위해 getWidth, getHeight 같은 함수를 이용해서 배치하고자 하였지만, 폰트의 특성 상 실제 크기와 보여지는 크기가 다름을 인지하게 되었습니다. FontMetrics라는 클래스를 몰랐기 때문에 x, y 값을 1 씩 조정하려다 보니 결과물이 정말 이상하게 나오게 되었습니다. 이후 FontMetrics를 사용하면 나름 간단하게 구현할 수 있다는 것을 알게 되었습니다.

두 번째는 비율 유지입니다. 비율을 유지하는 것이 어렵지 않을 것이라고 생각하였지만, 생각보다 더 많은 시간을 소요하였습니다. 처음 접근 방식은 모든 컴포넌트들의 비율을 수동으로 조정하는 것이었는데, 이는 불가능할 뿐만 아니라 너무 복잡한 코드를 낳게끔 한다는 것을 깨닫고 다른 방식을 모색해보았습니다. 결론적으로는 BorderLayout과 패딩을 잘 이용한다면 훨씬 수월하게 비율을 유지할 수 있고, BorderLayout을 사용하지 않는 경우에는 실제 사이즈를 조정할 수 있는 preferredSize를 조정하여 구현할 수 있다는 것을 깨달았습니다.

세 번째는 배치입니다. 분명히 setSize를 통해 사이즈를 조절했는데 적용이 안 되는 경우가 많았습니다. 이후에 Layout때문에 일어난 일임을 인지하게 되었습니다. 따라서 직접 사이즈를 조절하는 경우에는 preferredSize나 setLayout(null) 등을 이용해서 직접 배치할 수도 있지만, BorderLayout을 이용해서 padding 값을 조절하는 것이 가장 좋은 방법일 거라는 생각을 하게 되었습니다.

배운 점에 대해서도 생각해보았습니다. 처음에 계산기를 구성할 때 Stack을 이용해서 구성해야겠다고 생각했었지만, 막상 기능을 구현하면서 확인해보니 Stack을 굳이 사용하지 않아도 되겠다는 생각을 했고, 메모리적으로 봐도 그다지 효율적일 것이라고 생각하지 않게 되었습니다. 더하여 기능을 먼저 구현하기보다 배치나 색상을 먼저 생각하다보니 막상 기능 구현을 할 때 좋은 알고리즘을 짤 때 어려움을 겪은 것 같습니다. 다음에 이러한 프로그램을 구성하게 된다면, 기본적인 것들 생성을 제외하고 디자인적인 부분보다 기능적인 부분을 먼저 고려하여 더 나은 알고리즘이나 자료구조를 사용해서 프로그램을 개발해야겠다는 생각을 하게 되었습니다. 또한 Swing에도 정말 다양한 기능이나 메소드를 제공하고, 이를 잘 구성하기 위해 스케치 등의 과정도 필요한 것같다고 생각했습니다.

이번 과제를 진행하면서 정말 많은 시간을 소요했습니다. 하지만 일상 생활에서 사용되는 흔한 계산기와 같은 프로그램이 어떤 방식으로 Layout이나 배치, 기능을 구현하고 있는지 깨닫게 되었습니다. 과제를 풀면서 좋은 경험을 할 수 있었습니다. 감사합니다.

이상입니다.