Clean Code

3주차: 형식 맞추기 & 객체와 자료구조

서지혁

리디북스 스토어팀

2017년 5월 17일

형식 맞추기의 목적

형식 맞추기, 왜 하는가?

- because *important!*
- 형식 맞추기는 의사 소통을 위한 것
- 코딩 스타일과 가독성이 곧 지속 가능성과 확장 가능성
- 그래서, 형식을 어떻게 맞춰야 의사 소통을 잘 할 수 있을까?

형식 맞추기 (Formatting)

- 수직 포매팅
- 수평 포매팅
- '밥' 형님의 포매팅 규칙
- 팀 규칙

수직 포매팅 (Vertical Formatting)

- ‘신문’ 비유
- 수직 공간
- 수직 밀도
- 수직 거리
- 수직 순서

‘신문’ 비유

- 코드를 잘 짜여진 신문 기사처럼 짜자
- 파일명만으로도 우리가 보고 싶은 파일인지 알 수 있게
- 파일 시작 부분에는 고수준의 개념과 알고리즘
- 아래로 내려갈수록 디테일하게

수직 공간

- 거의 모든 코드는 왼쪽에서 오른쪽으로, 위에서 아래로 읽힌다
- 코드 한 줄 한 줄은 하나의 완전한 사고를 이루어야 함
- 각각의 이러한 사고들은 빈 줄로 분리되어야 함

수직 공간

Clean Code

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+?'''";
    private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
        Pattern.MULTILINE + Pattern.DOTALL
    );

    public BoldWidget(ParentWidget parent, String text) throws Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }

    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

수직 공간

Dirty Code

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+?'''";
    private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
        Pattern.MULTILINE + Pattern.DOTALL
    );
    public BoldWidget(ParentWidget parent, String text) throws Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }
    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

수직 밀도

- 수직 공간이 개념을 분리한다면
- 수직 밀도는 밀접한 관계를 다룬다
- 쓸 데 없는 내용으로 밀접해야 할 관계를 벌리지 말자

수직 밀도

Clean Code

```
public class ReporterConfig {
    private String m_className;
    private List<Property> m_properties = new ArrayList<Property>();

    public void addProperty(Property property) {
        m_properties.add(property);
    }
```

수직 밀도

Dirty Code

```
public class ReporterConfig {
    /**
     * The class name of the reporter listener
     */
    private String m_className;

    /**
     * The properties of the reporter listener
     */
    private List<Property> m_properties = new ArrayList<Property>();

    public void addProperty(Property property) {
        m_properties.add(property);
    }
```

수직 거리

- 코드의 의도를 읽을 시간에 코드의 위치를 찾게 하지 말자
- 연관된 코드는 수직 거리를 최소한으로, 서로 가까이 두자

수직 거리: 변수 선언

변수는 사용처와 최대한 가까이에 선언하기

```
private static void readPreferences() {
    InputStream is = null;
    try {
        is = new FileInputStream(getPreferencesFile());
        setPreferences(new Properties(getPreferences()));
        getPreferences().load(is);
    } catch (IOException e) {
        try {
            if (is != null)
                is.close();
        } catch (IOException e1) {
        }
    }
}
```

수직 거리: 변수 선언

루프 제어 변수는 루프 선언문 안에서 선언하기

```
public int countTestCases() {
    int count= 0;
    for (Test each : tests)
        count += each.countTestCases();
    return count;
}
```

수직 거리: 인스턴스 변수

한 장소에 모아 두자

```
public class TestSuite implements Test {
    static public Test createTest(Class<? extends TestCase> theClass,
            String name) {
        ...
    }
    public static Constructor<? extends TestCase>
    getTestConstructor(Class<? extends TestCase> theClass)
    throws NoSuchMethodException {
        ...
    }
    public static Test warning(final String message) {
        ...
    }
    private static String exceptionToString(Throwable t) {
        ...
    }
    private String fName;
    private Vector<Test> fTests= new Vector<Test>(10);
    public TestSuite() {
        ...
    }
    public TestSuite(final Class<? extends TestCase> theClass) {
        ...
    }
    public TestSuite(Class<? extends TestCase> theClass, String name) {
        ...
    }
    ... ... ... ... ...
}
```

수직 거리: 의존 함수

호출되는 함수는 호출하는 함수 아래 가까이

```
public class WikiPageResponder implements SecureResponder {
    ...
    public Response makeResponse(FitNesseContext context, Request request)
    throws Exception {
        String pageName = getPageNameOrDefault(request, "FrontPage");
        loadPage(pageName, context);
        if (page == null)
            return notFoundResponse(context, request);
        else
            return makePageResponse(context);
    }
    private String getPageNameOrDefault(Request request, String defaultPageName) {
        String pageName = request.getResource();
        if (StringUtil.isBlank(pageName))
            pageName = defaultPageName;
        return pageName;
    }
```

수직 거리: 개념적 유사성

유사한 개념일수록 수직 거리를 가까이

```
public class Assert {
    static public void assertTrue(String message, boolean condition) {
        if (!condition)
            fail(message);
    }
    static public void assertTrue(boolean condition) {
        assertTrue(null, condition);
    }
    static public void assertFalse(String message, boolean condition) {
        assertTrue(message, !condition);
    }
    static public void assertFalse(boolean condition) {
        assertFalse(null, condition);
    }
    ...
```

수직 순서

- 함수 호출 의존성이 아래로 향하게
- 호출되는 함수가 호출하는 함수 아래에
- 신문 기사처럼, 디테일을 아래쪽에 두자

수평 포매팅 (Horizontal Formatting)

- 수평 밀도
- 수평 정렬
- 들여쓰기
- 더미 스코프

수평 밀도

수평 밀도를 통해 개념의 밀접도를 구분

```
private void measureLine(String line) {
    lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize);
}
```

```
public class Quadratic {
    public static double root1(double a, double b, double c) {
        double determinant = determinant(a, b, c);
        return (-b + Math.sqrt(determinant)) / (2*a);
    }
    public static double root2(int a, int b, int c) {
        double determinant = determinant(a, b, c);
        return (-b - Math.sqrt(determinant)) / (2*a);
    }
    private static double determinant(double a, double b, double c) {
        return b*b - 4*a*c;
    }
}
```

수평 정렬

Dirty Code

```
public class FitNesseExpediter implements ResponseSender {
    private    Socket           socket;
    private    InputStream      input;
    private    OutputStream     output;
    private    Request          request;
    private    Response         reponse;
    private    FitNesseContext  context;
    protected long              requestParsingTimeLimit;
    private    long             requestProgress;
    private    long             requestParsingDeadline;

    public FitNesseExpediter(Socket          s,
                             FitNesseContext context) throws Exception
    {
        this.context             = context;
        socket                   = s;
        input                    = s.getInputStream();
        output                   = s.getOutputStream();
        requestParsingTimeLimit = 10000;
    }
```

수평 정렬

Clean Code

```
public class FitNesseExpediter implements ResponseSender {
    private Socket socket;
    private InputStream input;
    private OutputStream output;
    private Request request;
    private Response reponse;
    private FitNesseContext context;
    protected long requestParsingTimeLimit;
    private long requestProgress;
    private long requestParsingDeadline;

    public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception
    {
        this.context = context;
        socket = s;
        input = s.getInputStream();
        output = s.getOutputStream();
        requestParsingTimeLimit = 10000;
    }
```

들여쓰기

Dirty Code

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
```

들여쓰기

Clean Code

```
public class FitNesseServer implements SocketServer {
    private FitNesseContext context;
    public FitNesseServer(FitNesseContext context) {
        this.context = context;
    }
    public void serve(Socket s) {
        serve(s, 10000);
    }
    public void serve(Socket s, long requestTimeout) {
        try {
            FitNesseExpediter sender = new FitNesseExpediter(s, context);
            sender.setRequestParsingTimeLimit(requestTimeout);
            sender.start();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

더미 스코프

Clean Code

```
while (dis.read(buf, 0, readBufferSize) != -1)
    ;
```

- 가끔 while이나 for문의 내용이 더미일 때가 있음
- 웬만해선 이런 코드를 작성하는 건 피하자
- 꼭 필요할 땐 루프라는 사실을 명확히 하자

‘밥’ 형님의 포매팅 규칙

팀 규칙

- 누구나 자기만의 취향이 있지만
- 좋은 프로그래머 = 팀 플레이어
- 팀 단위의 규칙을 IDE 코드 포매터에 적용
- 가장 중요한 것은 일관성을 지키는 것

객체와 자료구조

- 구현 디테일이 노출되고, 그것에 의존하는 코드가 생기는 것을 피하기 위해 프라이빗 변수를 사용한다
- 그럼 대체 왜 객체에 getter와 setter를 만들어서 프라이빗 변수를 노출시키나요?
- 다음 페이지에! 공개됩니다

객체와 자료구조: 데이터 추상화

구현을 숨기는 건 추상화를 위한 것이다

```
public class Point {
    public double x;
    public double y;
}
```

```
public interface Point {
    double getX();
    double getY();
    void setCartesian(double x, double y);
    double getR();
    double getTheta();
    void setPolar(double r, double theta);
}
```

- 두 번째 예제가 아름다운 이유는 구현이 사각형인지 극좌표인지 모르지만, 분명 자료구조를 표현하기 때문
- 더 나아가, 자료구조 그 이상을 표현한다:
  - 접근 권한 설정
  - 각 좌표를 따로 읽을 수 있지만
  - 좌표 설정은 한꺼번에 같이 해야 한다
- 구현을 숨기는건 변수를 메서드로 숨기기 위한 것이 아니라 추상화를 위한 것

객체와 자료구조: 데이터/객체의 비대칭성

Procedural: 새 도형을 추가하고 싶다면 Geometry 클래스 함수를 모두 고쳐야 한다

```
public class Square {
    public Point topLeft;
    public double side;
}
public class Rectangle {
    public Point topLeft;
    public double height;
    public double width;
}
public class Circle {
    public Point center;
    public double radius;
}
public class Geometry {
    public final double PI = 3.141592653589793;
    public double area(Object shape) throws NoSuchShapeException {
        if (shape instanceof Square) {
            Square s = (Square)shape;
            return s.side * s.side;
        }
        else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle)shape;
            return r.height * r.width;
        }
        else if (shape instanceof Circle) {
            Circle c = (Circle)shape;
            return PI * c.radius * c.radius;
        }
        throw new NoSuchShapeException();
    }
}
```

객체와 자료구조: 데이터/객체의 비대칭성

Polymorphic: 새 함수를 추가하고 싶다면 도형 클래스 전부를 고쳐야 한다

```
public class Square implements Shape {
    private Point topLeft;
    private double side;
    public double area() {
        return side*side;
    }
}
public class Rectangle implements Shape {
    private Point topLeft;
    private double height;
    private double width;
    public double area() {
        return height * width;
    }
}
public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;
    public double area() {
        return PI * radius * radius;
    }
}
```

객체와 자료구조: 데이터/객체의 비대칭성

- 객체는 데이터를 추상화 뒤에 숨기고, 데이터를 조작하기 위한 함수를 노출한다
- 자료구조는 데이터를 노출하고, 별다른 기능이 없다
- 객체와 자료구조는 서로를 보완하는 관계
- 두 개의 차이와 각각의 장단점을 아는 것이 중요

객체와 자료구조: 디미터의 법칙

모듈은 자신이 조작하는 객체의 내부 구현을 몰라야 한다

"클래스 C의 메소드 f는 다음 객체의 메소드만 호출해야 한다"

- 클래스 C
- f가 생성한 객체
- f 인수로 넘어온 객체
- C 인스턴스 변수에 저장된 객체

객체와 자료구조: 디미터의 법칙

모듈은 자신이 조작하는 객체의 내부 구현을 몰라야 한다

기차 충돌:

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

코드 나누기:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

디미터 법칙을 거론할 필요가 없는 코드:

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

반 객체 반 자료구조인 잡종 구조를 다룰 때 머리가 아파지기 시작한다...

객체와 자료구조: 디미터의 법칙

모듈은 자신이 조작하는 객체의 내부 구현을 몰라야 한다

모두 객체라면 어떨까?

구조 감추기:

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

or

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

추상화 수준을 뒤섞어 놓음:

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

Clean Code:

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

객체와 자료구조: 자료 전달 객체

- 공개 변수만 있고 함수가 없는 클래스
- 'Data Transfer Object' 로 불리기도 함
- 데이터베이스 통신이나 소켓 메시지를 파싱할 때 사용됨

객체와 자료 구조: 결론

- 객체는 행위를 드러내고, 데이터는 숨긴다
- 자료 구조는 데이터를 드러내고, 별다른 기능이 없다
- 객체에는 새로운 데이터 타입을 추가하기가 쉽다
- 자료 구조에는 새로운 기능을 추가하기가 쉽다
- 문제 해결을 위한 최적의 방식을 편견 없이 고르는 것이 중요