# Java Training
# Info handling task
# (Into to design patterns)

# Agenda

- Previous homework review
- Design patterns
- UML class diagram
- Info handling task
- Composite pattern
- Chain of Responsibility
- Interpreter
- Homework


# Design patterns

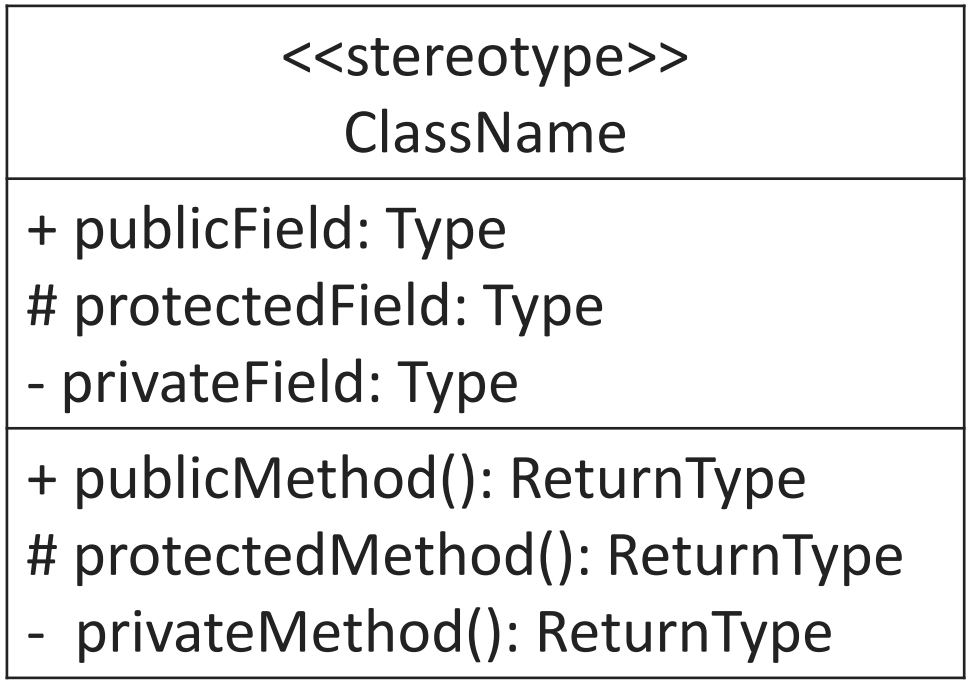
*Design pattern* is a general, reusable solution to a commonly occurring problem within a given context

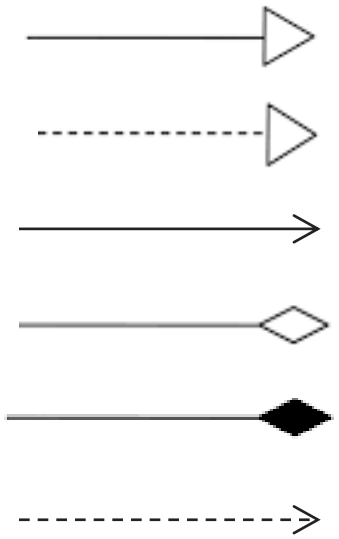
# Unified Modeling Language

# Unified Modeling Language

The *Unified Modeling Language (UML)* is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system

# Class Diagram in UML



Inheritance

Implementation

Association

Aggregation

Composition

Dependency

# Info handling task

*Implement an application that can parse text from file. Implement method that can calculate expressions in the text. Implement method that can build parsed text into String. The application also should be able to do three of the following operations:*

- Count sentences with equals words i.e. sentences that have two or more equal words
- Sort sentences by lexeme count in ascending or descending order
- In each sentence swap first and last lexeme
- Sort lexeme inside a sentence in alphabetical order
- Sort lexeme inside a sentence by letter count in ascending or descending order
- Remove all words of the given length
- Remove all words that starts from a given letter
- Reverse lexemes in sentences

* *lexeme* is either *word* or a *mathematical expression*

# Text sample

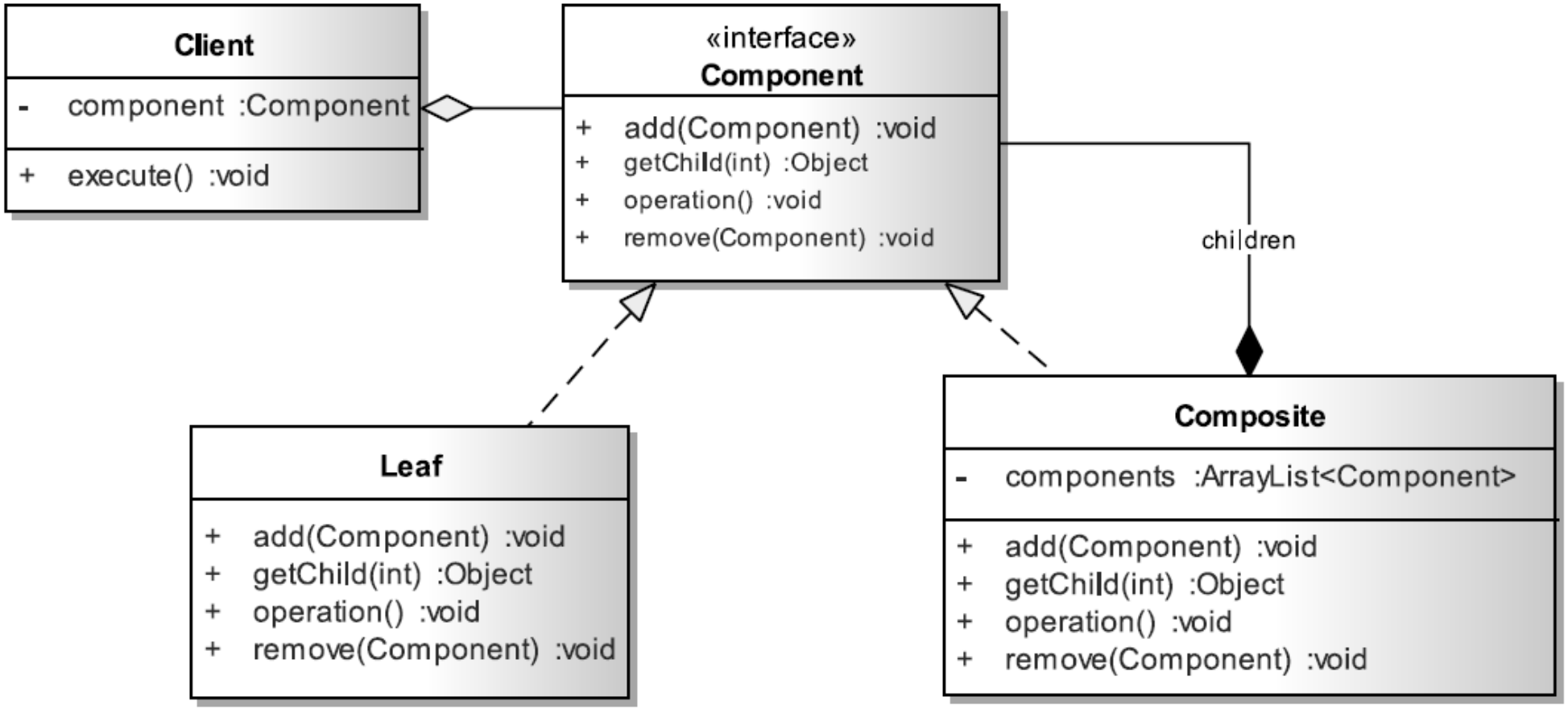
It has survived - not only (five) centuries, but also the leap into [13 *x* +] electronic typesetting, remaining [3 5 +] essentially [3 4 – 9 * 6 +] unchanged. It was popularised in the [ 20 1 - ] with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

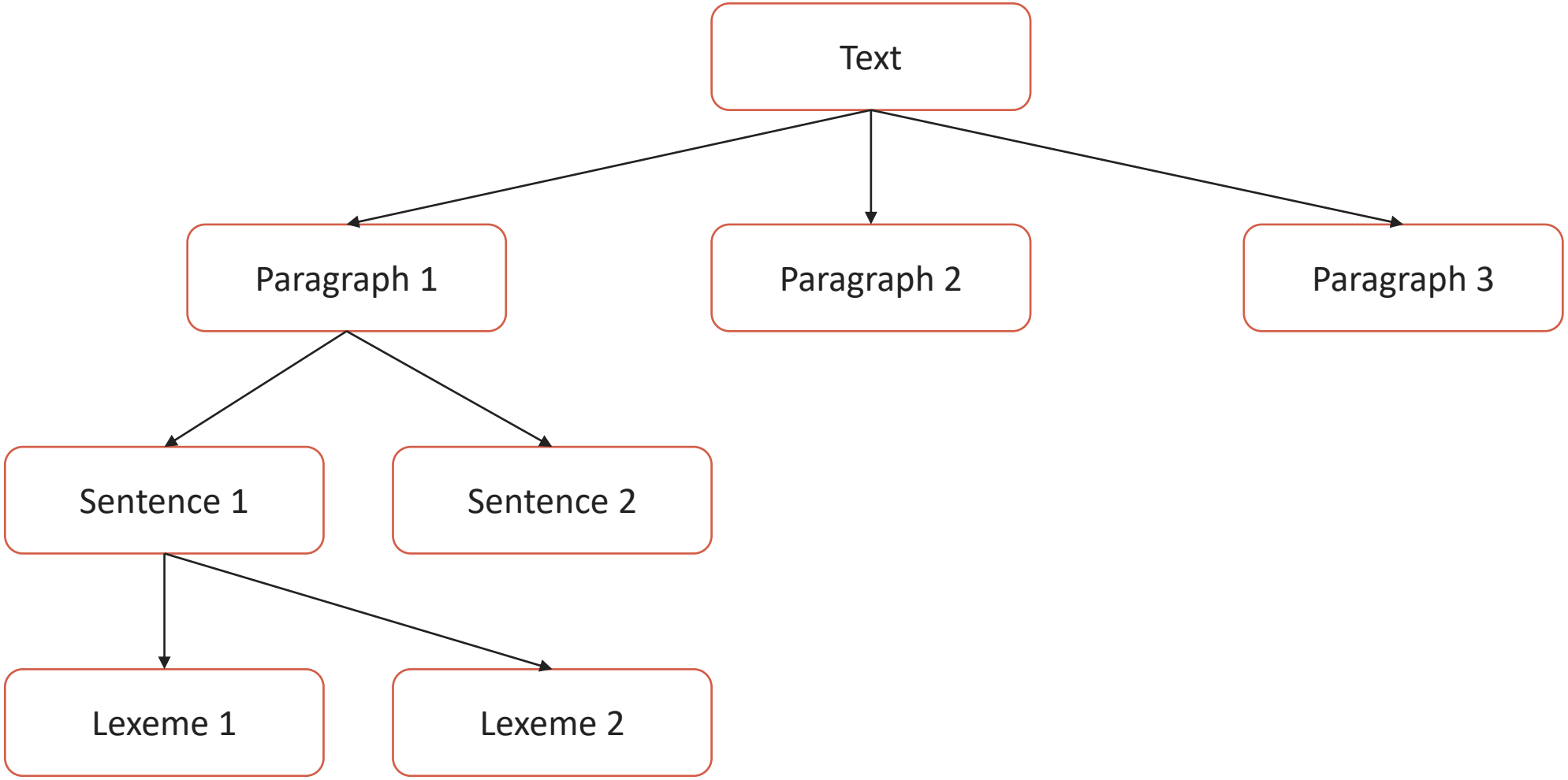
It is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout. The point of using [71 2 * 3 +] Ipsum is that it has a more-or-less normal distribution of letters, as opposed to using (Content here), content here', making it look like readable English.

It is a [5 *y* + 120 *] established fact that a reader will be of a page when looking at its layout.

# Composite pattern

# Composite pattern




Text
Paragraph 1
Paragraph 2
Paragraph 3
Sentence 1
Sentence 2
Lexeme 1
Lexeme 2

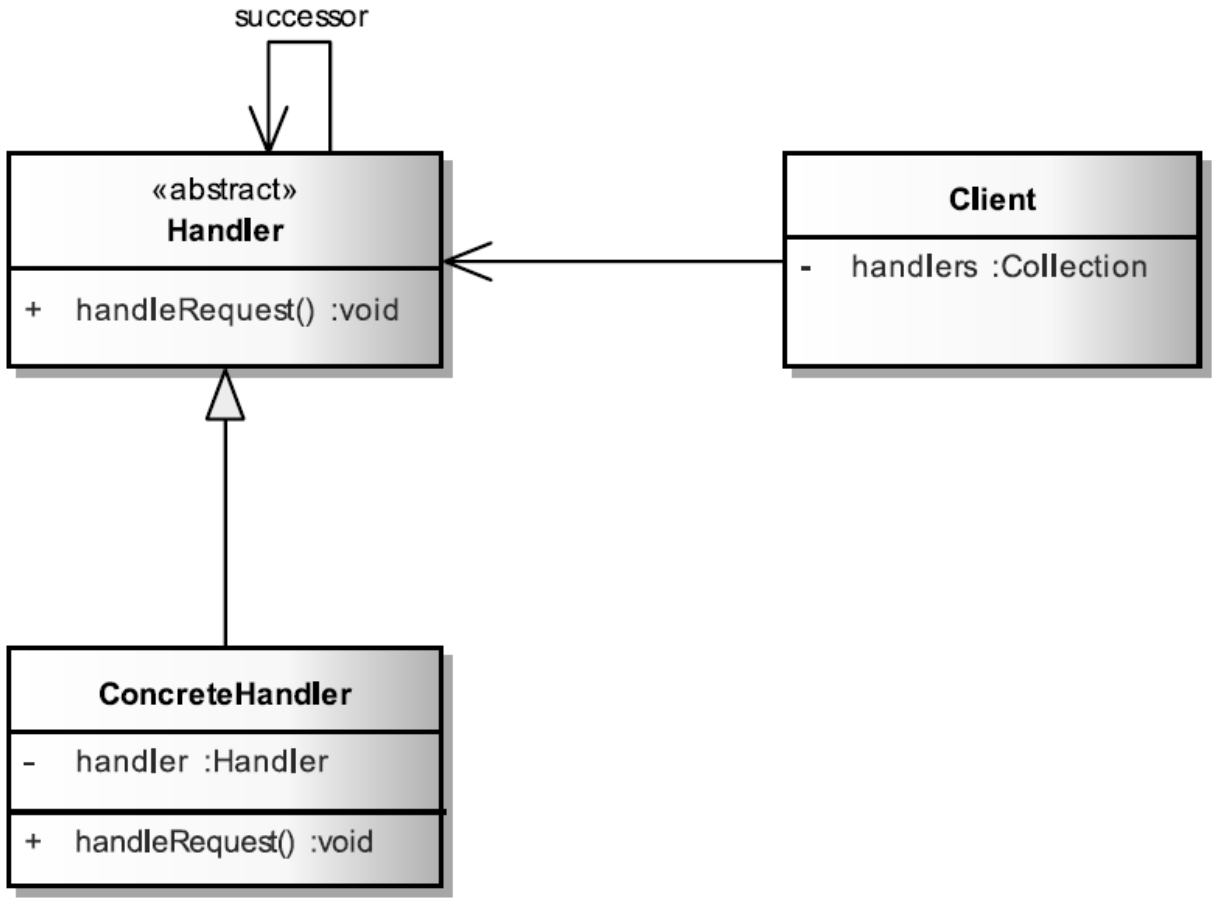
```
public interface Component {

    void operation();

    void add(Component c);

    void remove(Component c);

    Object getChild (int index);
}
```

```java
public class Composite implements Component {
    private List<Component> components = new ArrayList<Component>();

    @Override
    public void operation() {
        System.out.println("Composite -> Call children operations");
        int size = components.size();
        for (Component component : components) {
            component.operation();
        }
    }

    @Override
    public void add(Component component) {
        System.out.println("Composite -> Adding component");
        components.add(component);
    }

    @Override
    public void remove(Component component) {
        System.out.println("Composite -> Removing component");
        components.remove(component);
    }

    @Override
    public Object getChild(int index) {
        System.out.println("Composite -> Getting component");
        return components.get(index);
    }
}
```

```java
public class Leaf implements Component {

    @Override
    public void operation() {
        System.out.println("Leaf -> Performing operation");
    }

    @Override
    public void add(Component c) {
        System.out.println("Leaf -> add. Doing nothing");
        // generate an exception or return false if method is not 'void'
    }

    @Override
    public void remove(Component c) {
        System.out.println("Leaf -> remove. Doing nothing");
        // generate an exception or return false if method is not 'void'
    }

    public Object getChild(int index) {
        throw new UnsupportedOperationException();
    }
}
```

# Chain of Responsibility

# Chain of Responsibility



```java
public interface Parser {

    Composite parse(String text);

}
```

```java
public abstract class AbstractParser implements Parser {

    private Parser successor;

    AbstractParser(Parser successor) {
        this.successor = successor;
    }

    protected Parser getSuccessor() {
        return successor;
    }

}
```

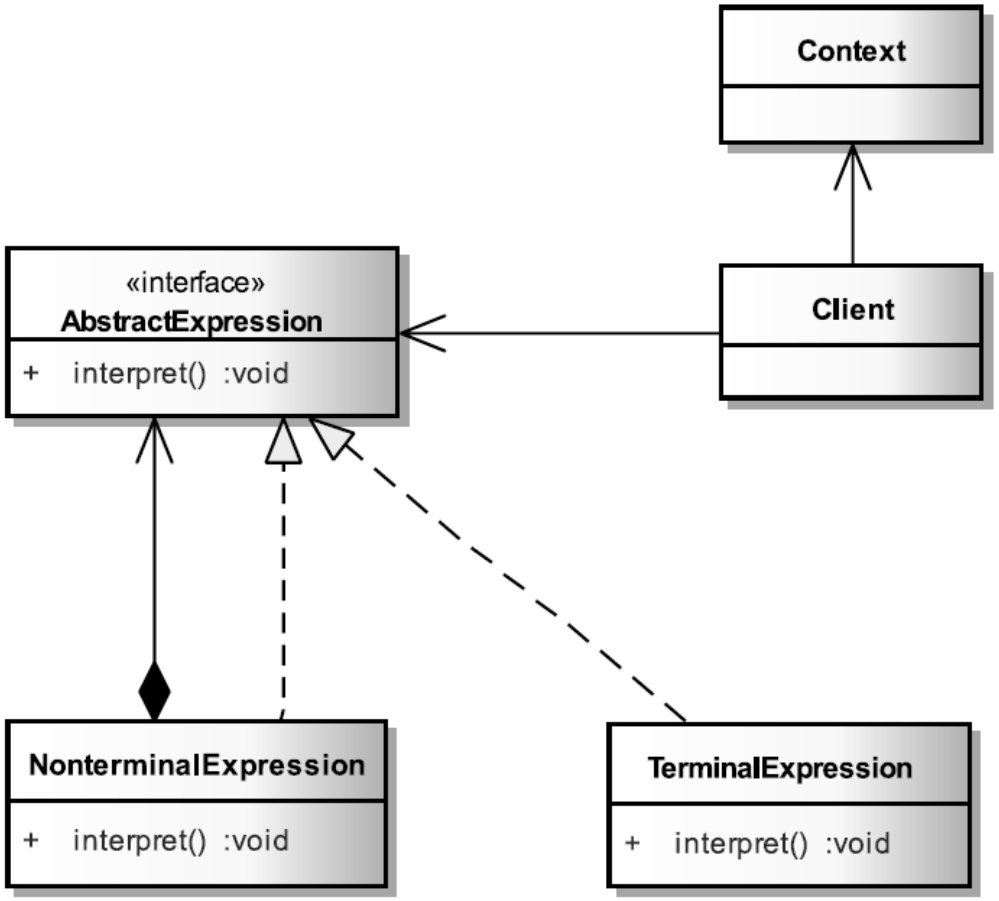
```java
public class TextParser extends AbstractParser {

    private static final String SPLITTER = "\n";

    public TextParser(Parser successor) {
        super(successor);
    }

    public Composite parse(String text) {
        Composite composite = new Composite();
        String[] parts = text.split(SPLITTER);
        for (String part : parts) {
            Composite inner = getSuccessor().parse(part);
            composite.add(inner);
        }
        return composite;
    }
}
```

```
public class ChainBuilder {

    public Parser build(){
        return new TextParser(new ParagraphParser(...));
    }
}
```

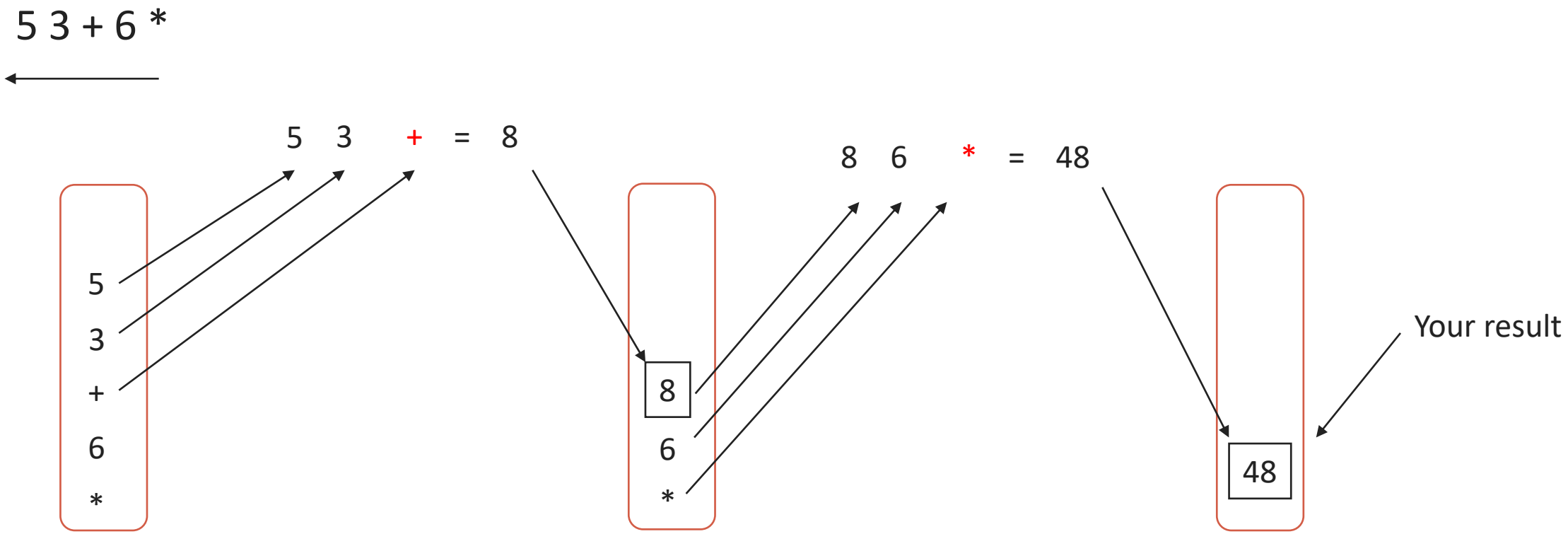
```
public class TextProcessor {

    public Composite parseText(String text) {
        Parser parser = new ChainBuilder().build();
        return parser.parse(text);
    }

}
```

# Interpreter

# Interpreter



# Expression calculation (postfix notation)



```java
public interface Expression {

    void interpret(Context context);
}


public class Context {

    private Deque<Integer> contextValues = new ArrayDeque<>();

    public Integer popValue() {
        return contextValues.pop();
    }
    public void pushValue(Integer value) {
        this.contextValues.push(value);
    }
}
```

```java
public class NonterminalExpression implements Expression {

    private int number;

    public NonterminalExpression(int number) {
        this.number = number;
    }

    public void interpret(Context context) {
        context.pushValue(number);
    }
}


public class TerminalDivideExpression implements Expression {

    public void interpret(Context context) {
        context.pushValue(context.popValue()/context.popValue());
    }
}
```

```
public class Calculator {
    private List<Expression> parse(String expression) {
        List<Expression> expressions = new ArrayList<>();
        for (String lexeme : expression.split("\\s+")) {
            if (lexeme.isEmpty()) {
                continue;
            }
            char temp = lexeme.charAt(0);
            switch (temp) {
                case '+':
                    expressions.add(new TerminalPlusExpression());
                    break;
                case '-':
                    expressions.add(new TerminalMinusExpression());
                    break;
                case '*':
                    expressions.add(new TerminalMultiplyExpression());
                    break;
                case '/':
                    expressions.add(new TerminalDivideExpression());
                    break;
                default:
                    Scanner scan = new Scanner(lexeme);
                    if (scan.hasNextInt()) {
                        expressions.add(
                                new NonterminalExpression(scan.nextInt()));
                    }
            }
        }
        return expressions;
    }
```

```
    public int calculate(String expression) {
        Context context = new Context();
        List<Expression> expressions = parse(expression);
        for (Expression terminal : expressions) {
            terminal.interpret(context);
        }
        return context.popValue();
    }
```

# Text Logic

```
public class TextLogic {

    public Composite calculate(Composite text) {
        ...
    }

    public Composite reverse(Composite text) {
        ...
    }

    public String restore(Composite text) {
        ...
    }
}
```

# Homework requirements

- The text should be parsed into an object. This object should be a tree containing paragraphs, sentences, lexemes. The lexeme is either word or math expression. Use Composite pattern
- Model classes should have no logic
- You should be able to restore text from the object. Multiple spaces or table could be one space after restoration
- Use regular expression to parse the text. In your code regular expressions should be constants.
- Use Chain of Responsibility when parsing the text
- Your parsers should be stateless, do not create unnecessary parser object, ideally you should have only one parser object (do not use singleton pattern).
- You should be able to calculate math expressions . Использовать Interpreter.
- Use Log4J2 for logging.
- Should implements 3 additional logical operations on the text (see slides in the beginning)
- No main class should be added. Use unit tests

# Homework template

- Fork from repo 

- To task according to description and project structure
- Write tests

https://github.com/filippstankevich/infohandling

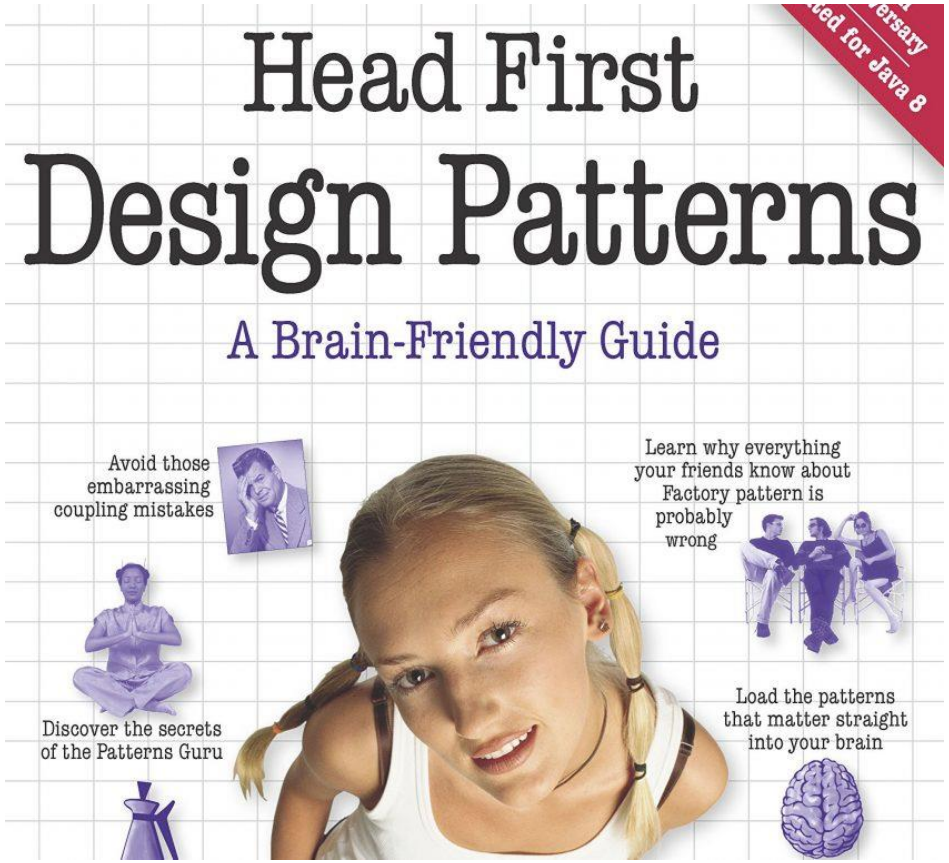
https://refactoring.guru/smells/long-method

# Useful links

- https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/
- https://sourcemaking.com/design_patterns

# Books

- Head First Design Patterns: A Brain-Friendly Guide



# Thanks