FUNDAMENTAL PROGRAMMING TECHNIQUES

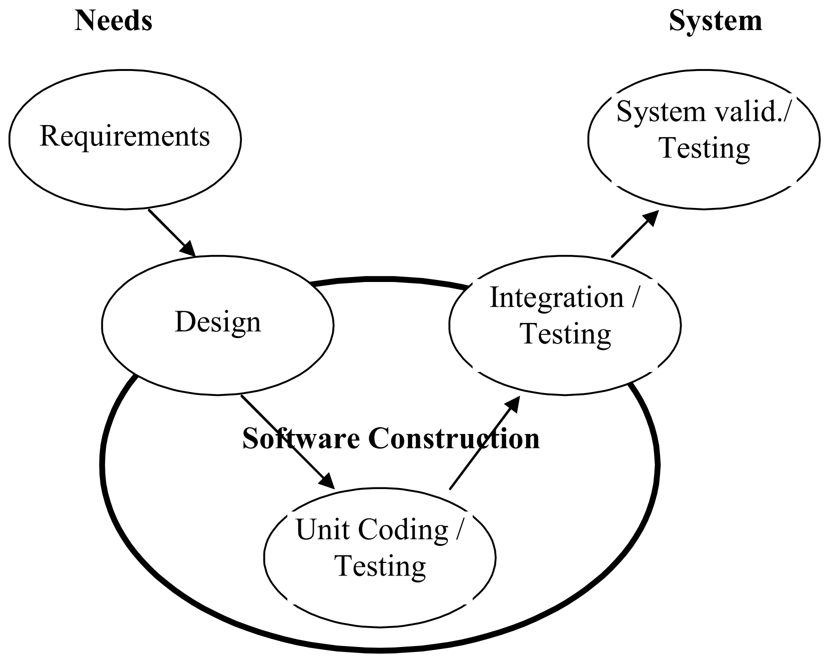
ASSIGNMENT 1 – SUPPORT PRESENTATION

Outline

- Software development process in the context of Assignment 1
  - Problem and Solution
  - Objectives
  - Analysis
  - Design
  - Unit Testing with Junit

- Theoretical Background
  - Basics of polynomial arithmetic
  - Graphical User Interfaces Development using Swing
  - Regular expressions and pattern matching

Software Development Process



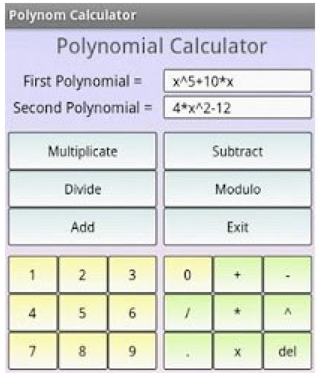
Problem and solution

PROBLEM: “Performing polynomial operations on paper is difficult and time consuming.”

$$\begin{array}{r} 2x^2 + 3x + 1 \\ + \quad x^3 + 6x^2 - 5 \\ \hline x^3 + (2x^2 + 6x^2) + 3x + 1 - 5 \end{array}$$

How to design and implement the solution?

SOLUTION: Polynomial calculator

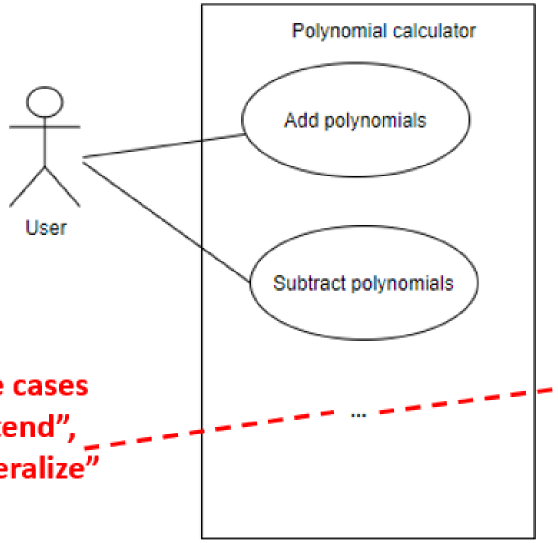


1. Clearly state the main objective and the sub-objectives required to reach it.
2. Analyze the problem and define the functional and non-functional requirements.
3. Design the solution
4. Implement the solution
5. Test the solution

Objectives

- Main objective
  - Design and implement a polynomial calculator with a dedicated graphical interface through which the user can insert polynomials, select the mathematical operation to be performed and view the result.
- Sub-objectives
  - Analyze the problem and identify requirements
  - Design the polynomial calculator
  - Implement the polynomial calculator
  - Test the polynomial calculator

Analysis



Can you identify use cases connected with "extend", "include" and "generalize" relationships?

What other use cases can you identify?

Use Case: add polynomials
Primary Actor: user
Main Success Scenario:

1. The user inserts the 2 polynomials in the graphical user interface.
2. The user selects the "addition" operation
3. The user clicks on the "compute" button
4. The polynomial calculator performs the addition of the two polynomials and displays the result

Alternative Sequence: Incorrect polynomials

- The user inserts incorrect polynomials (e.g. with 2 or more variables)
- The scenario returns to step 1

Define requirements

Functional requirements:

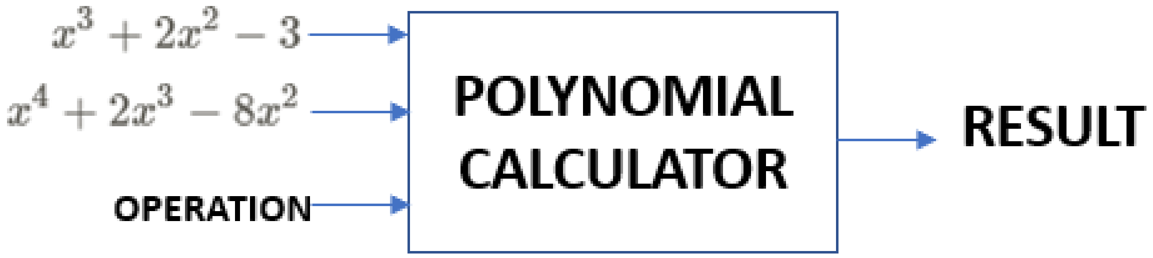
- The polynomial calculator should allow users to insert polynomials
- The polynomial calculator should allow users to select the mathematical operation
- The polynomial calculator should add two polynomials
- … what other functional requirements can you define? …

Non-Functional requirements:

- The polynomial calculator should be intuitive and easy to use by the user
- … what other non-functional requirements can you define? …

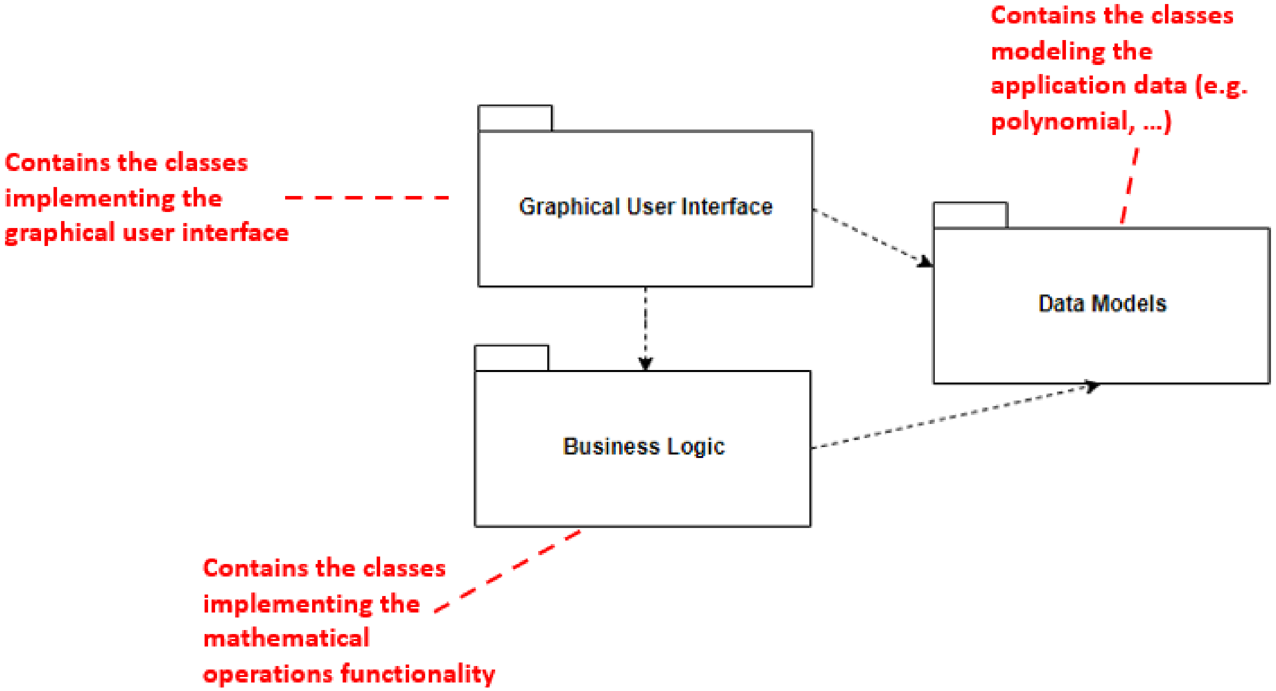
Design

Level 1: Overall system design



Design

Level 2: Division into sub-systems/packages



GOOD PRACTICE - Use architectural patterns
Architectural patterns define structures for software systems in terms of predefined subsystems and their responsibilities. Structural patterns (e.g. Layers) and interactive systems patterns (e.g. **Model View Controller**) are some examples of architectural patterns types.

Design
Level 2: Division into sub-systems/packages

- Model View Controller Architectural Pattern [Ref]

Context

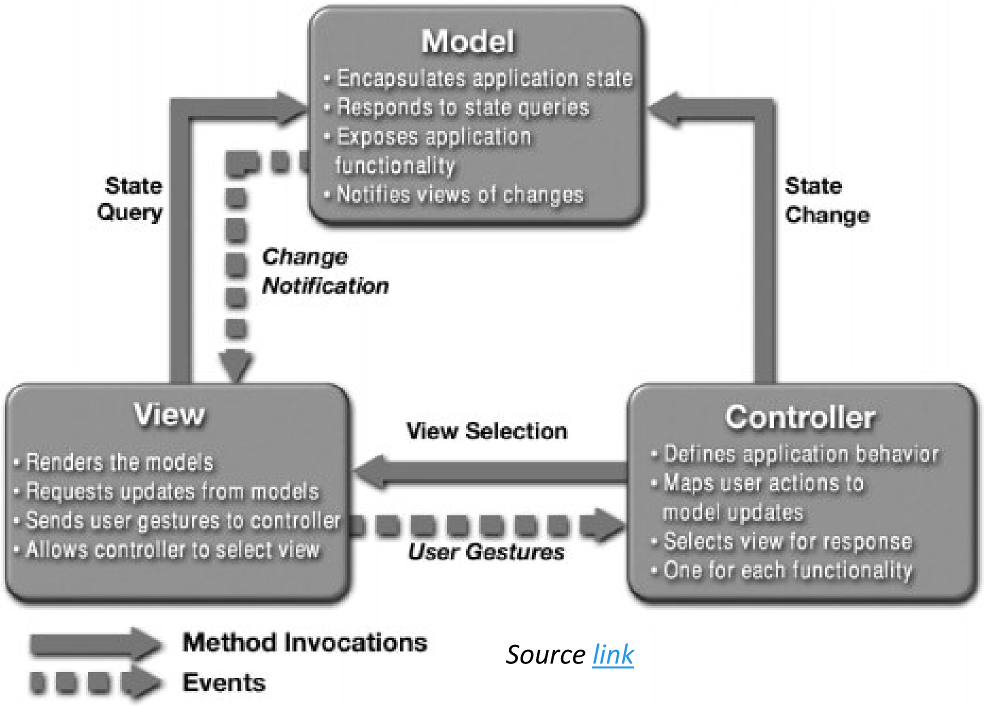
- Many software systems deal with finding data from a repository and displaying the data to the users through a graphical user interface (GUI)
  - Users can modify the data and the modifications are saved back in the repository
  - Continuous information flow between the GUI and the repository => might be tempted to implement everything in the same class
  - ⇒ **Disadvantages**:
    - ⇒ The GUI changes more often than the business logic implementation -> if they are implemented in the same class then each time the GUI changes the business logic is changed
    - ⇒ The business logic can not be reused
    - ⇒ The code is complex and difficult to maintain

Solution

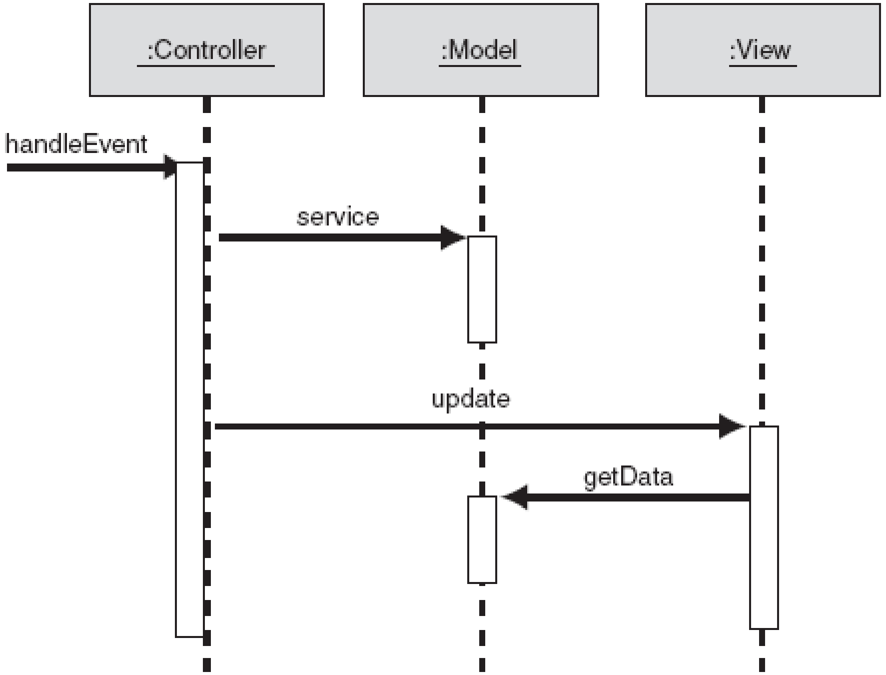
- Use the Model View Controller pattern which divides the application into three areas: processing, output and input
  - **Model components** – encapsulate core data and functionality
  - **View components** – display information to the user - obtains the data it displays from the model
  - **Controller** – Each view has an associated controller component. Controllers receive input, usually as events that denote mouse movement, activation of mouse buttons or keyboard input. Events are translated to service requests, which are sent either to the model or to the view.

Design
Level 2: Division into sub-systems/packages

- Model View Controller Architectural Pattern

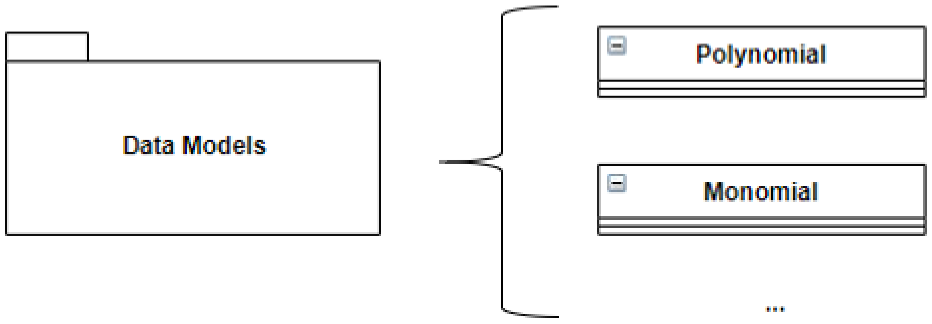


Source link



Check this link for an example of applying the Model View Controller Pattern

Design
Level 3: Division into classes



- **What is the relation between polynomial and monomial?**
- **Which are the attributes of the Polynomial and Monomial classes?**
- **Which are the public methods of the Polynomial and Monomial classes?**

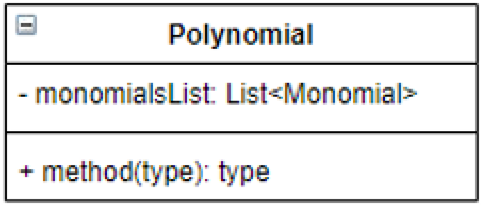
...

How do we devise the other packages in classes?

! When defining the classes think about **ABSTRACTION, INHERITANCE,** and **ENCAPSULATION**

Design

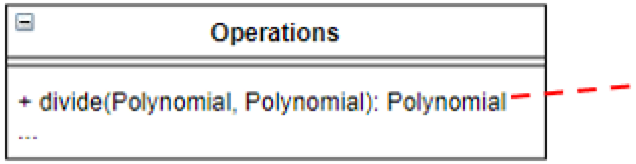
Level 4: Division into routines (i.e. methods)



What private methods should we define for the Polynomial class?

Design

Level 5: Internal routine design




```
function n / d is
    require d ≠ 0
    q ← 0
    r ← n             // At each step n = d × q + r

    while r ≠ 0 and degree(r) ≥ degree(d) do
        t ← lead(r) / lead(d)       // Divide the leading terms
        q ← q + t
        r ← r − t × d

    return (q, r)
```

- **Implementation…**

Unit Testing with JUnit

- Software testing
  - The process of executing a piece of software to identify any gaps, errors, or missing requirements in contrary to the actual requirements
- Software test
  - Piece of software which validates whether the execution of another piece of software
    - Results in the expected state (state testing) – result validation
    - Is done according to the expected sequence of events (behavior testing)
- Types of testing
  - Unit testing – targets small units of code
  - Integration testing
- Testing frameworks for Java
  - Junit, TestNG

Source [link]()

Unit Testing with JUnit

- Configure Maven to work with Junit – add the Junit dependency in pom.xml

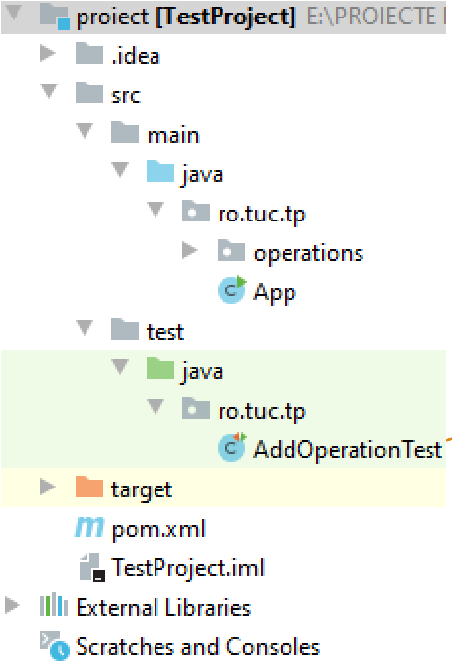
```xml
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.1.0</version>
  <scope>test</scope>
</dependency>
```

- Consider the class *AddOperation* that defines a single method for adding two numbers

```java
public class AddOperation {
    public int add(int a, int b){
        return a + b;
    }
}
```

Unit Testing with JUnit

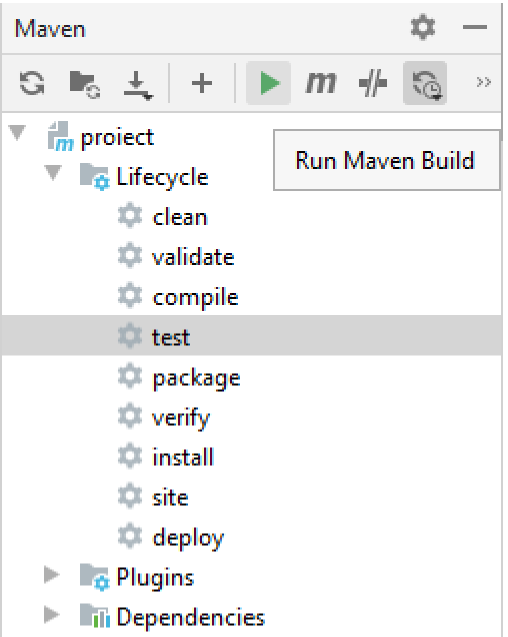
- Create the test class
  - Create a java test class named *AddOperationTest.java* and place it in src/main/test
  - Implement a test method named *addTest* in your test class
  - Specify the annotation @Test to the method *addTest*()
  - Implement the test condition and check the condition using *assertEquals* API of JUnit

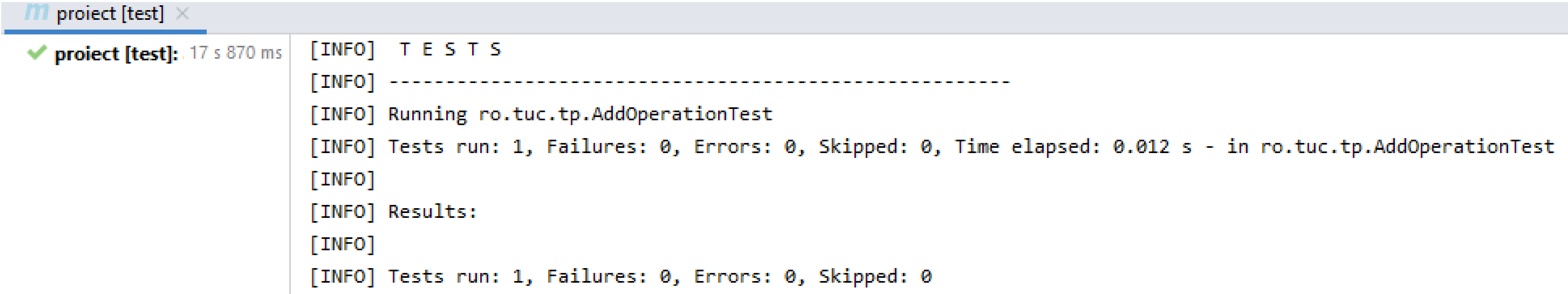



```java
import org.junit.jupiter.api.Test;
import ro.tuc.tp.operations.AddOperation;

import static org.junit.jupiter.api.Assertions.assertTrue;

public class AddOperationTest {
    @Test
    public void addTest(){
        AddOperation addOperation = new AddOperation();
        assertTrue( condition: addOperation.add( a: 4,  b: 5) == 9,
                 message: "The result of adding 4 and 5 is 9");
    }
}
```

Unit Testing with JUnit

- Run the test





Unit Testing with JUnit

- **Basic Annotations** (Link)

Annotation	Description
@Test	Denotes that a method is a test method.
@ParameterizedTest	Denotes that a method is a parameterized test.
@RepeatedTest	Denotes that a method is a test template for a repeated test.
@BeforeEach	Denotes that the annotated method should be executed before each @Test, @RepeatedTest, @ParameterizedTest, method in the current class.
@AfterEach	Denotes that the annotated method should be executed after each @Test, @RepeatedTest, @ParameterizedTest method in the current class.
@BeforeAll	Denotes that the annotated method should be executed before all @Test, @RepeatedTest, @ParameterizedTest methods in the current class;
@AfterAll	Denotes that the annotated method should be executed after all @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class.
…	…

Unit Testing with JUnit

- **Assertions** are static methods defined in the org.junit.jupiter.api.Assertions class: assertEquals, assertAll, assertNotEquals, assertTrue, etc. - check (Link) for more examples
  - In case the assertion facilities provided by JUnit Jupiter are not sufficient enough, third party libraries can be used (e.g. AssertJ, Hamcrest, etc.)

- **Test suites** – aggregate multiple test classes in a suite so that they can be run together

```
@RunWith(JUnitPlatform.class)
@SelectPackages("ro.tuc.tp.example")
public class AllUnitTest {
}
```

```
@RunWith(JUnitPlatform.class)
@SelectClasses({AddOperationTest.class})
public class AllUnitTest {
}
```

Note: in order to run test suites add dependencies for junit-platform-runner, junit-4.13, junit-jupiter-api and junit-jupiter-engine

Unit Testing with JUnit

- **Parameterized Tests** [Link] - make it possible to run a test multiple times with different arguments
  - Must declare at least one source that will provide the arguments for each invocation and then consume the arguments in the test method

```java
public class AParameterizedTest {
    private AddOperation addOperation = new AddOperation();

    @ParameterizedTest
    @MethodSource("provideInput")
    void testAdditions(int a, int b, int expectedResult){
        assertEquals(expectedResult, addOperation.add(a, b));
    }

    private static List<Arguments> provideInput(){
        List<Arguments> argumentsList=new ArrayList<>();
        argumentsList.add(Arguments.of(2, 3, 5));
        argumentsList.add(Arguments.of(4, 7, 11));
        argumentsList.add(Arguments.of(10, 3, 13));
        return argumentsList;
    }
}
```

Note:

1) Add the following dependency

```xml
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-params</artifactId>
  <version>5.4.2</version>
  <scope>test</scope>
</dependency>
```

2) The method providing the arguments must be static

Theoretical Background

Basics of polynomial arithmetic

A *polynomial P* in an indeterminate *X* is formally defined as:

$$P(X) = a_n * X^n + a_{n-1} * X^{n-1} + \cdots + a_1 * X + \ a_0$$

where:

- $a_1, a_2, \ldots, a_n$ represent the polynomial's coefficients
- n represents the polynomial degree

A *monomial* is a special type of polynomial with only one term.

Consider another *polynomial Q* in the indeterminate *X* which is formally defined as:

$$Q(X) = b_n * X^n + b_{n-1} * X^{n-1} + \cdots + b_1 * X + \ b_0$$

Basics of polynomial arithmetic

Addition of two polynomials:

$$P(X) + Q(X) = (a_n + b_n) * X^n + (a_{n-1} + b_{n-1}) * X^{n-1} + \cdots + (a_1 + b_1) * X + (a_0 + b_0)$$

Example:

Consider the following two polynomials:

$$P(X) = 4 * X^5 - 3 * X^4 + X^2 - 8 * X + 1$$

$$Q(X) = 3 * X^4 - X^3 + X^2 + 2 * X - 1$$

The result of adding the two polynomials is:

$$P(X) + Q(X) = 4 * X^5 - X^3 + 2 * X^2 - 6 * X$$

Basics of polynomial arithmetic

Subtraction of two polynomials:

$$P(X) - Q(X) = (a_n - b_n) * X^n + (a_{n-1} - b_{n-1}) * X^{n-1} + \cdots + (a_1 - b_1) * X + (a_0 - b_0)$$

Example:

Consider the following two polynomials:

$$P(X) = 4 * X^5 - 3 * X^4 + X^2 - 8 * X + 1$$

$$Q(X) = 3 * X^4 - X^3 + X^2 + 2 * X - 1$$

The result of subtracting the polynomials is:

$$P(X) - Q(X) = 4 * X^5 - 6 * X^4 + X^3 - 10 * X + 2$$

Basics of polynomial arithmetic

Multiplication of two polynomials

To multiply two polynomials, multiply each monomial in one polynomial by each monomial in the other polynomial, add the results and simplify if necessary.

Example: Consider the following two polynomials:

$$P(X) = 3 * X^2 - X + 1$$

$$Q(X) = X - 2$$

The result of multiplying the two polynomials is:

$$P(X) * Q(X) = 3 * X^3 - X^2 + X - 6 * X^2 + 2 * X - 2 = 3 * X^3 - 7 * X^2 + 3 * X - 2$$

Basics of polynomial arithmetic

Division of two polynomials

To divide two polynomials *P* and *Q*, the following steps should be performed:

Step 1 - Order the monomials of the two polynomials *P* and *Q* in descending order according to their degree.

Step 2 - Divide the polynomial with the highest degree to the other polynomial having a lower degree (let's consider that *P* has the highest degree)

Step 3 – Divide the first monomial of *P* to the first monomial of *Q* and obtain the first term of the quotient

Step 4 – Multiply the quotient with *Q* and subtract the result of the multiplication from *P* obtaining the remainder of the division

Step 5 – Repeat the procedure from step 2 considering the remainder as the new dividend of the division, until the degree of the remainder is lower than *Q*.

Example: Consider the following two polynomials:

$P(X) = X^3 - 2 * X^2 + 6 * X - 5$

$Q(X) = X^2 - 1$

The result of dividing the two polynomials is:

$$
\begin{array}{l}
\underline{(X^3 - 2*X^2 + 6*X - 5)} : (X^2 - 1) = X - 2 \\
\underline{-X^3 \qquad\quad + \; X} \\
\quad - 2*X^2 + 7*X - 5 \\
\quad \underline{2*X^2 \qquad\quad - 2} \\
\qquad\qquad 7*X - 7
\end{array}
$$

Quotient = X – 2; Remainder = 7*X-7

Basics of polynomial arithmetic

Derivative of a polynomial

The derivative of a polynomial P is defined as follows:

$$\frac{d}{dx}(a_n * X^n + a_{n-1} * X^{n-1} + \cdots + a_1 * X + a_0) = n * a_n * X^{n-1} + (n-1) * a_{n-1} * X^{n-2} + \cdots + a_1$$

Example: Consider the following polynomial:

$$P(X) = X^3 - 2 * X^2 + 6 * X - 5$$

The derivative of polynomial P is:

$$\frac{d}{dx}(X^3 - 2 * X^2 + 6 * X - 5) = 3 * X^2 - 4 * X + 6$$

Basics of polynomial arithmetic

Integral of polynomials

The integral of a polynomial P is defined as follows:

$$\int a_n * X^n + a_{n-1} * X^{n-1} + \cdots + a_1 * X + a_0 = \int a_n * X^n dx + \int a_{n-1} * X^{n-1} dx + \cdots + \int a_1 * X dx + \int a_0 dx$$

where:

$$\int a_n * X^n dx = \frac{a_n * X^{n+1}}{n+1} + C$$

Example: Consider the following polynomial:

$$P(X) = X^3 + 4 * X^2 + 5$$

The integral of polynomial P is computed as:

$$\int P(X) dx = \int X^3 + 4 * X^2 + 5 = \int X^3 dx + \int 4 * X^2 dx + \int 5 dx = \frac{X^{3+1}}{3+1} + \frac{4 * X^{2+1}}{2+1} + \frac{5 * X^{0+1}}{0+1} + C = \frac{X^4}{4} + \frac{4 * X^3}{3} + 5 * X + C$$

Source link

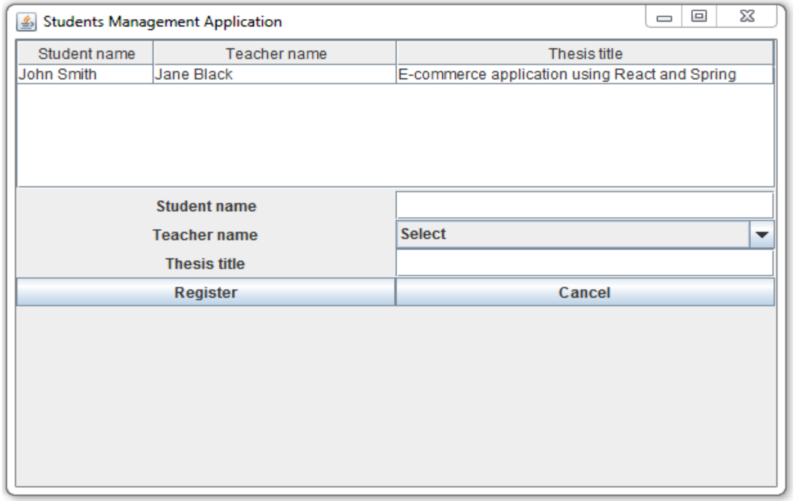
Graphical User Interfaces Development using Swing

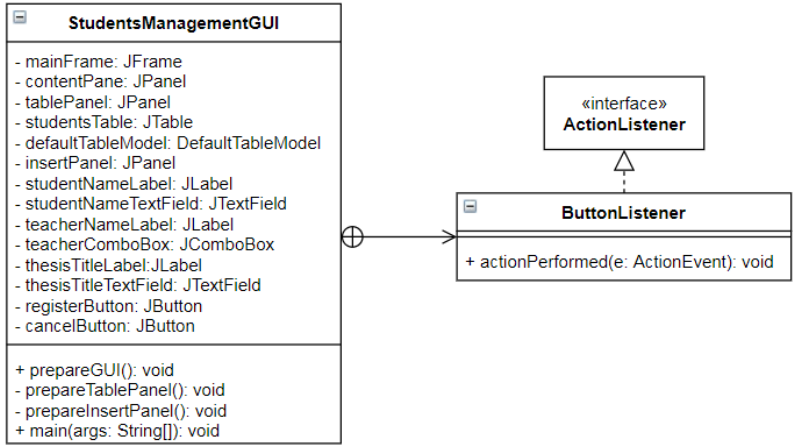
- SWING API [Link]
  - Is part of the Java Foundation Classes (JFC)
  - Offers facilities to write applications with a graphical user interface
  - Includes 17 packages consisting of classes and interfaces
- **javax.swing** - Is the most important package from Swing

Component Type	Examples
Atomic components	JLabel, JButton, JCheckBox, JRadioButton, JToggleButton, JScrollBar, JSlider
Complex components	JTable, JTree, JComboBox, JList, JFileChooser, JColorChooser, JOptionPane
Text components	JTextField, JPasswordField, JTextArea, JEditorPane, JTextPane
Menus	JMenuBar, JMenu, JPopupMenu, JMenuItem, CheckboxMenuItem , JRadioButtonMenuItem
Intermediate containers	JPanel, JTabbedPane, JDesktopPane
Top level containers	JFrame, JDialog

Graphical User Interfaces Development using Swing

- Example – students management application





GOOD TO KNOW – TOP-LEVEL CONTAINERS [Link]

The graphical components must be included in a containment hierarchy having a top-level container (e.g. JFrame, JDialog) as root. In particular, the graphical components will be contained in the content pane of the top-level container. A menu bar can be included in a top-level container, but it will reside outside the content pane. To create and set up a frame, the following steps should be performed:

- Create the frame by instantiating the *JFrame* class.
- Create components and add them to the frame's content pane.
- Size the frame manually (using the *setSize* method), or automatically (using the pack method).
- Show the frame onscreen (using the *setVisible* method).

To get the content pane of a JFrame component, the method getContentPane defined in the *JFrame* class is used. There are 2 approaches for setting the content pane of a JFrame component:

1) Use the method *getContentPane*() defined in the *JFrame* class to get the frame's content pane and add various components to it: **mainFrame.getContentPane().add(tablePanel);**

Note: mainframe.add(tablePanel) can also be used as the add method has been overridden and it actually adds tablePanel to the frame's content pane

2) Use the JFrame's setContentPane method to make another component the content pane of the frame:

```
JPanel contentPanePanel = new JPanel();
// add other graphical components to contentPanePane
  …
mainFrame.setContentPane(contentPanePanel);
```

Graphical User Interfaces Development using Swing

- Example – students management application

```
private void prepareGUI() {
        mainFrame = new JFrame("Students Management Application");
        mainFrame.setSize(500, 500);
        mainFrame.addWindowListener(new WindowAdapter() {
             @Override
             public void windowClosing(WindowEvent e) {
                  System.exit(0);
             }
          });
        contentPane = new JPanel(new GridLayout(2, 1));
        prepareTablePanel();
        prepareInsertPanel();
        mainFrame.setContentPane(contentPane);
        mainFrame.setVisible(true);
}
```

The *setSize* method is used to explicitly set the size of the frame.

Adds a window listener to receive window events from the *JFrame* component. In this case, once the frame is closed, the application is exited.

GridLayout is used as the *Layout Manager of* the *contentPane JPanel* which serves as the content pane of the frame. *GridLayout* origanizes the graphical components on rows and columns, setting the same size for all components.

The *setVisible(true)* method is used to make the frame appear onscreen.

The *contentPane JPanel* component is set as the content pane of the frame. All other graphical components will be contained in it.

```
private void prepareTablePanel(){
        tablePanel = new JPanel();
        tablePanel.setLayout(new BoxLayout(tablePanel, BoxLayout.PAGE_AXIS));
        defaultTableModel = new DefaultTableModel();
        defaultTableModel.addColumn("Student name");
        defaultTableModel.addColumn("Teacher name");
        defaultTableModel.addColumn("Thesis title");
        defaultTableModel.addRow(new Object[] {"John Smith", "Jane Black", "E-
                        commerce application using React and Spring"});
        studentsTable = new JTable(defaultTableModel);
        JScrollPane scrollPane = new JScrollPane(studentsTable);
        studentsTable.setFillsViewportHeight(true);
        this.tablePanel.add(scrollPane);
        contentPane.add(tablePanel);
    }
```

BoxLayout is used as the Layout Manager of tablePanel in order to organize the contained graphical components on top of each other.

A *DefaultTableModel* object uses a Vector of Vectors to store the cell value objects of a *JTable* component.

The *JScrollPane* object is used as a container for the *JTable Component* and automatically places the table's header at the top and they remain visible even when the table data is scrolled. If a *JScrollPane* is not used, then the table's header must be manually placed in the container.

The *tablePanel* is added to the content pane of the frame.

Graphical User Interfaces Development using Swing

- Layout Managers are used to organize graphical components in containers. The following Layout Managers can be used [Link]:

a) BorderLayout – places the components in 5 areas: top, bottom, left, right, and centre.

b) BoxLayout – places the components on a row or on a column.

c) CardLayout – enables the implementation of an area that contains different components at different times.

d) FlowLayout – places the components in a single row.

e) GridBagLayout – places the components in a grid of cells, allowing the spanning and sizing of components over multiple cells.

f) GridLayout – sets equal sizes for the components and places them in the requested number of rows and columns.

Regular expressions and pattern matching

- **java.util.regex package** [Ref]
  - Contains classes used for pattern matching with regular expressions
  - Regular expression = sequence of characters defining a search pattern
  - Result of matching a regular expression against a text
    - True/false result -> specifies if the regular expression matched the text
    - Set of matches – one match for every occurrence of the regular expression found in the text
  - Consists of the classes:

Class	Description
Pattern	• **Pattern object** = compiled representation of a regular expression • **compile() methods** - accept a regular expression as the first argument, to return a **Pattern object**

Matcher	• **Matcher object** = engine that interprets the pattern and performs match operations against an input string
	• **matcher() method** – invoked on a Pattern object to obtain a Matcher object
	• Other methods
	  • **Index methods** (start, end) – show where the match was found in the input string
	  • **Study methods** (lookingAt, find, matches) – review the input string and return a Boolean indicating whether or not the pattern is found
	  • **Replacement methods** (appendReplacement, appendTail, replaceAll, replaceFirst, quoteReplacement) – replace text in an input string

PatternSyntaxException	**PatternSyntaxException object** – unchecked exception indicating syntax error in a regular expression pattern

Regular expressions and pattern matching

Category	Construct	Matches
Character classes	[abc]	a, b, or c (simple class)
	[^abc]	Any character except a, b, or c (negation)
	[a-zA-Z]	a through z or A through Z, inclusive (range)
	[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
	[a-z&&[def]]	d, e, or f (intersection)
	[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
	[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)
Predefined character classes	.	Any character
	\d	A digit: [0-9]
	\D	A non-digit: [^0-9]
	\s	A whitespace character: [\t\n\x0B\f\r]
	\S	A non-whitespace character: [^\s]
	\w	A word character: [a-zA-Z_0-9]
	\W	A non-word character: [^\w]
Logical operators	*XY*	*X* followed by *Y*
	X|*Y*	Either *X* or *Y*
	(*X*)	X, as a capturing group

Category	Construct	Matches
Greedy quantifiers	*X*?	*X*, once or not at all
	*X**	*X*, zero or more times
	X+	*X*, one or more times
	X{*n*}	*X*, exactly *n* times
	X{*n*,}	*X*, at least *n* times
	X{*n*,*m*}	*X*, at least *n* but not more than *m* times
Reluctant quantifiers	*X*??	*X*, once or not at all
	*X**?	*X*, zero or more times
	X+?	*X*, one or more times
	X{*n*}?	*X*, exactly *n* times
	X{*n*,}?	*X*, at least *n* times
	X{*n*,*m*}?	*X*, at least *n* but not more than *m* times
Possessive quantifiers	*X*?+	*X*, once or not at all
	*X**+	*X*, zero or more times
	X++	*X*, one or more times
	X{*n*}+	*X*, exactly *n* times
	X{*n*,}+	*X*, at least *n* times
	X{*n*,*m*}+	*X*, at least *n* but not more than *m* times

Quantifiers are used to specify the number of occurrences to match against – at first glance it may appear that they do exactly the same thing but there are subtle implementation differences between them:

1) **Greedy quantifiers** force the matcher to read in, or eat, the entire input string prior to attempting the first match. If the first match attempt (the entire input string) fails, the matcher backs off the input string by one character and tries again, repeating the process until a match is found or there are no more characters left to back off from.

```
Enter your regex: .*foo  // greedy quantifier
Enter input string to search: xfooxxxxxxfoo
I found the text "xfooxxxxxxfoo" starting at index 0 and ending at index 13.
```

2) **Reluctant quantifiers** start at the beginning of the input string, then reluctantly eat one character at a time looking for a match. The last thing they try is the entire input string.

```
Enter your regex: .*?foo  // reluctant quantifier
Enter input string to search: xfooxxxxxxfoo
I found the text "xfoo" starting at index 0 and ending at index 4.
I found the text "xxxxxxfoo" starting at index 4 and ending at index 13.
```

3) **Possessive quantifiers** always eat the entire input string, trying once (and only once) for a match.

```
Enter your regex: .*+foo // possessive quantifier
Enter input string to search: xfooxxxxxxfoo
No match found.
```

Source [link]

Regular expressions and pattern matching

- **Example -** Create a regular expression for validating Romanian mobile phone numbers. A valid mobile phone number should contain 10 digits, out of which the first 2 should be 07, and the rest from 0 to 9

```
…
String PHONE_PATTERN = "07[0-9]{8}";
String PHONE_EXAMPLE = "1711123456";
Pattern pattern = Pattern.compile(PHONE_PATTERN);
Matcher matcher = pattern.matcher(PHONE_EXAMPLE);
if(matcher.matches()){
     System.out.println("The phone is valid");
}
else {
     System.out.println("The phone is not valid");
}
…
```

To test your regular expressions check this link

Regular expressions and pattern matching

- **Capturing groups**
  - Are a way to treat multiple characters as a single unit
  - Are created by placing the characters to be grouped inside a set of parentheses – example: (ABC)
  - Are numbered by counting their opening parenthesis from left to right – check the example below

The **expression ((A)(B(C)))** contains 4 groups

Group number	Matching
1	((A)(B(C)))
2	(A)
3	(B(C))
4	(C)

- **Example**

```
String text = "John writes about this, and John Doe writes about that," +
                        " and John Wayne writes about everything.“;
String patternString1 = "((John) (.+?)) ";
Pattern pattern = Pattern.compile(patternString1);
Matcher matcher = pattern.matcher(text);
while(matcher.find()) {
    System.out.println("found: <"  + matcher.group(1) +
                    "> <"       + matcher.group(2) +
                    "> <"       + matcher.group(3) + ">");
        }
```

```
found: <John writes> <John> <writes>
found: <John Doe> <John> <Doe>
found: <John Wayne> <John> <Wayne>
```

Sources link1 and link2