Modelowanie i Analiza Systemów informacyjnych (MAS)

dr inż. Mariusz Trzaska
mtrzaska@pjwstk.edu.pl

Wykład 13

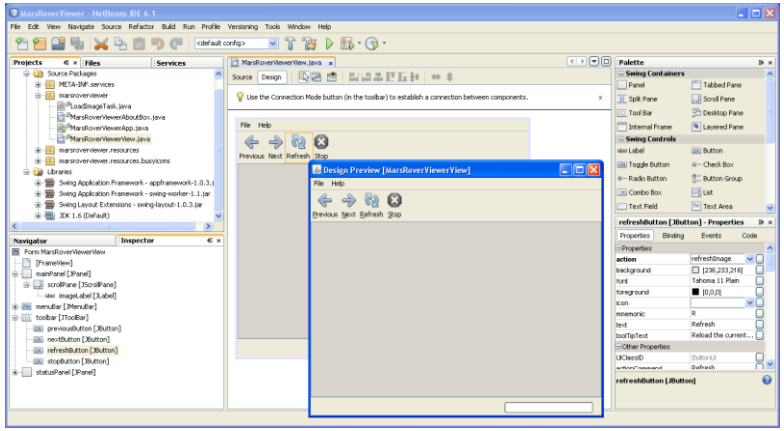
Tworzenie graficznych interfejsów użytkownika (2)

Zagadnienia

- Wstęp
- Tworzenie prostego GUI z wykorzystaniem edytora NetBeans.
- Długotrwałe akcje, a GUI
  - Problemy,
  - Rozwiązanie problemów przy pomocy wątków.
- Tworzenie rozbudowanej aplikacji.
- Zastosowanie edytora GUI przy implementacji aplikacji MDI.

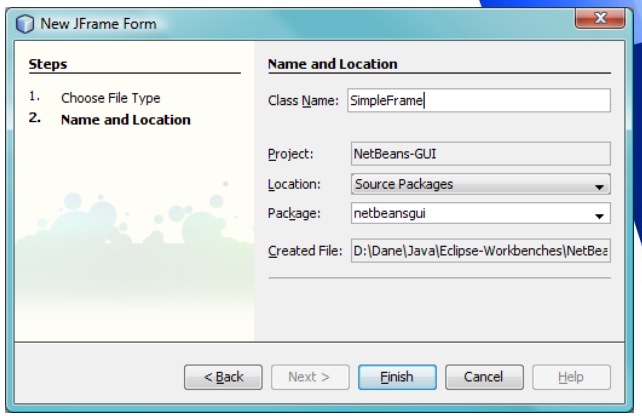
Edytor GUI – NetBeans

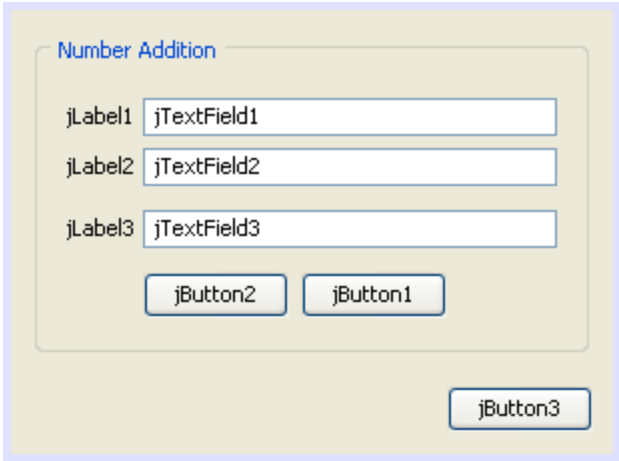
- Całkowicie graficzne projektowanie GUI,
- Bardzo dobry system wspierający pozycjonowanie kontrolek (*layout manager*),
- Wiele różnych komponentów,
- Obsługa Beans Binding technology (JSR 295)
- Obsługa Swing Application Framework (JSR 296)



Wykorzystanie edytora GUI – prosty przykład

- Nowy projekt (Sumator): *File > New Project: Java/Java Application*
- Tworzymy nowe okno (*JFrame*): węzeł projektu i menu *New/JFrame Form*
- Dodajemy *JPanel* (razem z odpowiednim *Border*)
- Dodajemy
  - etykiety (*Label*),
  - pola tekstowe (*Text Field*)
  - przyciski (*Button*).





http://www.netbeans.org/kb/60/java/gui-functionality.html

Wykorzystanie edytora GUI – prosty przykład (2)

- Zmieniamy nazwy pól tekstowych oraz przycisków:
  - Zaznaczamy kontrolkę
  - Z menu kontekstowego wybieramy *Change Variable Name* i podajemy nową nazwę (np. *txtFirstNumber* lub *btnAdd*).
- Zmieniamy opisy etykiet oraz przycisków
  - Podwójne kliknięcie na etykiecie tekstowej i podanie nowej nazwy,
  - Kliknięcie na przycisku i zmiana własności *Text* w oknie *Properties*.

http://www.netbeans.org/kb/60/java/gui-functionality.html

Wykorzystanie edytora GUI – prosty przykład (3)

- Dodajemy funkcjonalność
  - Musimy wiedzieć kiedy zostanie wciśnięty któryś z przycisków. W tym celu użyjemy odpowiedniego *Listener'a* nasłuchującego zdarzeń związanych z konkretną kontrolką, np. przyciskiem.
  - Zaznaczamy kontrolkę (np. przycisk *Exit*) i z menu kontekstowego wybieramy: *Events/Action/actionPerformed*. IDE wygenerowało nam procedurę obsługi do której musimy wstawić odpowiedni kod.
  - W procedurze umieszczamy wołanie metody „biznesowej", która wykona odpowiednie działania. Dzięki umieszczeniu metody, a nie bezpośrednio kodu możemy ją użyć również w innych miejscach programu.

.../gui-functionality.html

Wykorzystanie edytora GUI – prosty przykład (4)

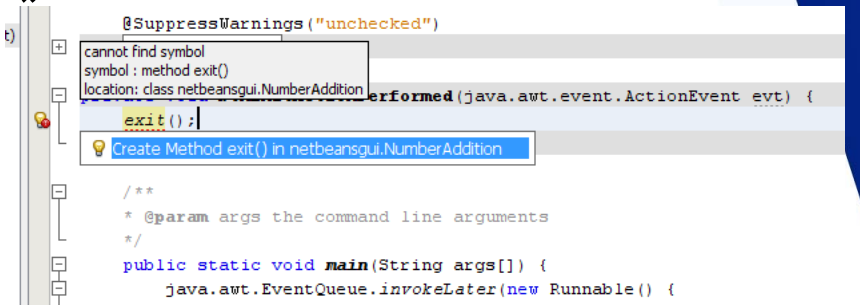
- Dodajemy funkcjonalność – c.d.
  - Dla przycisku *Exit* podajemy nazwę metody kończącej działanie aplikacji, np. *exit()*. Ponieważ taka metoda nie istnieje, IDE

    poinformuje nas

    o błędzie oraz

    zaproponuje stworzenie

    takiej metody.



  - Dodajemy niezbędny komentarz

    JavaDoc wpisując

    */*** i wciskając *Enter*.

```
/**
 * Exits the application.
 */
private void exit() {
    System.exit(0);
}
```

  - Procedurę powtarzamy dla wszystkich przycisków.

http://www.netbeans.org/kb/60/java/gui-functionality.html

Wykorzystanie edytora GUI – prosty przykład (5)

- Dodajemy funkcjonalność – c.d.

```
/**
 * Clears the input and output fields.
 */
private void clearInput() {
    txtFirstNumber.setText("");
    txtSecondNumber.setText("");
    txtResult.setText("");
}

/**
 * Counts the result.
 */
private void count() {
    float num1, num2, result;

    // Convert the text into number
    var txtNumber1 = txtFirstNumber.getText();
    num1 = (txtNumber1 == null || txtNumber1.isEmpty()) ? 0 : Float.parseFloat(txtNumber1);
    var txtNumber2 = txtSecondNumber.getText();
    num2 = (txtNumber2 == null || txtNumber2.isEmpty()) ? 0 : Float.parseFloat(txtNumber2);

    // Count the result
    result = num1 + num2;

    // Put the result into the field
    txtResult.setText(String.valueOf(result));
}
```

http://www.netbeans.org/kb/60/java/gui-functionality.html

Wykorzystanie edytora GUI – prosty przykład (6)

- Uruchamiamy nasz program, wybierając opcję *Run file* z menu kontekstowego odpowiedniego pliku projektu.
- Jeżeli zobaczymy komunikat o braku metody `main` dodajemy poniższy kod:

```
public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {

        public void run() {
            new NumberAddition().setVisible(true);
        }
    });
}
```

http://www.netbeans.org/kb/60/java/gui-functionality.html

Wykorzystanie edytora GUI – prosty przykład (7)

Demonstracja tworzenia prostego GUI w NetBeans

Długotrwałe akcje, a GUI

- Użytkownicy nie lubią gdy komputer nie reaguje na ich polecenia.
- Może się tak dziać, gdy aplikacja wykonuje jakieś długotrwałe akcje, np.
  - wyszukiwanie elementów,
  - obliczenia,
  - przetwarzanie danych,
  - pobieranie/wysyłanie zasobów z/do sieci,
  - oczekiwanie na dostępność usług.

Długotrwałe akcje, a GUI (2)

- W takiej sytuacji należy:
  - poinformować użytkownika o trwających czynnościach,
  - uaktualniać na bieżąco informacje o ich postępie (prawdziwą lub „udawaną”),
  - umożliwić ich przerwanie.
- Badania pokazują, że nawet długo trwające czynności, gdy towarzyszy im odpowiednie GUI, wydają się krócej trwać i mniej irytują.
- Jeżeli jesteśmy w stanie przewidzieć, że jakaś czynność może długo trwać, nie zaszkodzi też uzyskać potwierdzenie od użytkownika dotyczące jej rozpoczęcia.

Długotrwałe akcje, a GUI (3)

- Dlaczego poniższy kod nie zadziała zgodnie z naszymi oczekiwaniami?

```
Public void longWorkingMethod() {

    For(...) {
        // Executing a long lasting method,
        // e.g. looking for an information

        // Updating GUI
        progressBar.setProgress(...);
    }
}
```

?!

- Na czym polega problem?
- Jak go rozwiązać?

Długotrwałe akcje, a GUI (4)

- Niestety współczesne języki programowania nie obsługują takich sytuacji bezpośrednio.
- W celu prawidłowej realizacji długotrwałej akcji, programista musi użyć **wątki**.
- Zwykle do tego typu implementacji warto zastosować klasę pomocniczą dostarczaną razem z językiem:
  - `SwingWorker` dla Javy (kiedyś nie była oficjalną częścią dystrybucji),
  - `BackgroundWorker` dla MS C#,
  - `javafx.concurrent.Task` dla JavaFX (*JavaDoc*).

Długotrwałe akcje, a GUI (5)

- Wspomniane klasy są zbudowane przy wykorzystaniu podobnej koncepcji. Programista musi przesłonić metodę:
  - wykonująca długotrwałą akcję: `SwingWorker.doInBackground()`,
  - wykonywaną po zakończeniu akcji: `SwingWorker.done()`.
- Oprócz tego jest dostępnych kilka innych metod umożliwiających, m.in.
  - przerwanie operacji,
  - uzyskanie częściowych wyników.

SwingWorker - przykłady

- W minimalnej wersji musimy przesłonić metodę wykonująca długotrwałą czynność.
- Klasa `SwingWorker` jest parametryzowana:
  - Typ wartości zwracanej przez metody (rezultat długotrwałej akcji):
    - `doInBackground()`,
    - `get()`. Metoda oczekują na zakończenie całej akcji i zwraca wartośc dopiero wtedy (uwaga: takie oczekiwanie blokuje GUI).
  - Typ wartości przekazywanych jako wyniki cząstkowe (w postaci listy).

SwingWorker – przykłady (2)

- Prosty przykład obliczający sumę liczb od 1 do *maxNumber*.

```
public class Worker extends SwingWorker<Integer, Void> {

    @Override
    /**
     * Count the sum of numbers from 1 to the given maxNumber and return it as
String.
     */
    protected Integer doInBackground() throws Exception {
        // Count the sum of numbers from 1 to the given maxNumber
        final int maxNumber = 500000000;
        int sum = 0;

        for (long i = 1; i < maxNumber; i++) {
            sum += i;
        }

        return sum;
    }
}
```

SwingWorker – przykłady (3)

- Przykład użycia.

```
public static void main(String[] args) {
    Worker worker = new Worker();

    System.out.println("Counting...");

    // Start the long-lasting action
    worker.execute();

    // But the control flow returns immediately
    System.out.println("Waiting for the result...");

    // Exceptions required by SwingWorker
    try {
        // Waiting and checking if it is finished
        while(!worker.isDone()) {
            Thread.sleep(1000);
        }

        // We have the result - get it
        var sum = worker.get();

        System.out.println("The result: " + sum);
    } catch (InterruptedException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null,
ex);
    } catch (ExecutionException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null,
ex);
    }
}
```

SwingWorker – przykłady (4)

- Przykład również obliczający sumę liczb od 1 do *maxNumber*, ale z informowaniem o postępie obliczeń (konsola).

```
public class WorkerPlus extends SwingWorker<Integer, Void> {

    /**
     * Count the sum of numbers from 1 to the given maxNumber with updates.
     */
    @Override
    protected Integer doInBackground() throws Exception {
        // Count the sum of numbers from 1 to the given maxNumber
        final int maxNumber = 500000000;
        int sum = 0;

        for (int i = 1; i < maxNumber; i++) {
           sum += i;
           if(i % 100 == 0) {
                // Store the progress info
                double percentage = 100.0 * (float) i / maxNumber;
                setProgress((int) percentage);
            }
        }

        return sum;
    }
}
```

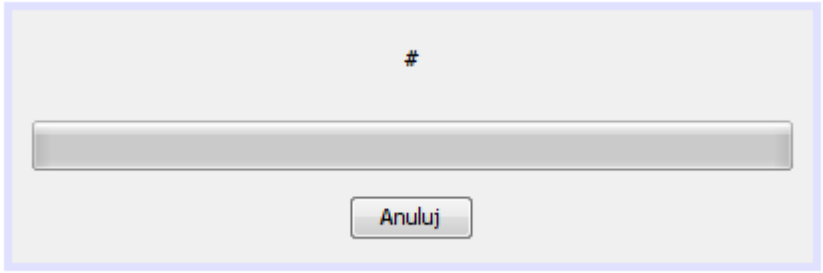
SwingWorker – przykłady (5)

- Przykład użycia.

```
public static void main(String[] args) {
    var worker = new WorkerPlus();

    System.out.println("Counting...");

    // Start the long-lasting action
    worker.execute();

    // But the control flow returns immediately
    System.out.println("Waiting for the result...");


    // Exceptions required by SwingWorker
    try {
        // Waiting and checking if it is finished
        while(!worker.isDone()) {
            System.out.println("Progress: " + worker.getProgress());
            Thread.sleep(100);
        }

        // We have the result - get it
        int floatSum = worker.get();

        System.out.println("The result: " + String.valueOf(floatSum));
    } catch (InterruptedException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    } catch (ExecutionException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

SwingWorker – przykład z GUI

- Tworzymy odpowiednie okno (`JDialog`) w edytorze GUI
  - `JLabel` wyświetlający komunikat,
  - `JProgressBar` z graficzną informacją o postępie,
  - `JButton` umożliwiający anulowanie operacji,
  - Kilka pomocniczych metod.



SwingWorker – przykład z GUI (2)

- Tworzymy odpowiednie okno (JDialog) w edytorze GUI – c.d.

```
public class ProgressDialog extends javax.swing.JDialog {
    private SwingWorker worker;

    /**
     * Sets a worker connected to the dialog.
     * @param worker
     */
    public void setWorker(SwingWorker worker) {
        this.worker = worker;
    }

    /**
     * Creates new form ProgressDialog
     */
    public ProgressDialog(java.awt.Frame parent, String message) {
        super(parent, true);
        initComponents();

        lblMessage.setText(message);
    }

    public void setValue(Integer progress) {
        progressBar.setValue(progress);
    }

    private void initComponents() {
        // [...]
    }
}
```

SwingWorker – przykład z GUI (3)

- Rozszerzamy klasę `SwingWorker` implementując naszą długotrwałą czynność.

```
public class NumberGenerator extends SwingWorker<List<Integer>,
                                                  Void> {
    private List<Integer> numbers;
    private int maxCount;
    ProgressDialog progressDialog;

    public NumberGenerator(int maxCount, ProgressDialog
                           progressDialog) {
        super();

        this.maxCount = maxCount;
        this.progressDialog = progressDialog;
    }

    // [...]

    @Override
    protected void done() {
        super.done();

        // Hide the GUI (dialog window)
        progressDialog.setVisible(false);
    }
}
```

SwingWorker – przykład z GUI (4)

- Rozszerzamy klasę `SwingWorker` implementując naszą długotrwałą czynność – c.d.

```
public class NumberGenerator extends SwingWorker<List<Integer>, Void> {
    private List<Integer> numbers;
    private int maxCount;
    ProgressDialog progressDialog;

    // [...]
    /** Generates the numbers and returns them as a list. */
    @Override
    protected List<Integer> doInBackground() throws Exception {
        final int maxNumber = 1000;
        int i = 1;

        numbers = new LinkedList<Integer>();

        // Check if we already generated all required numbers all the operation has been
cancelled
        while(i < maxCount && isCancelled() == false) {
            Integer curInt = new Integer((int)(Math.random() * (double) maxNumber));
            numbers.add(curInt);

            if(i % 100 == 0) {
                // Store the update info
                setProgress((int)(100.0 * (float)i / maxCount));
            }

            i++;
        }
        return numbers;    }}
```

SwingWorker – przykład z GUI (5)

- Tworzymy okno umożliwiające uruchomienie generowania.

```
public class MainFrame extends javax.swing.JFrame {

   public MainFrame() {
      initComponents();
   }

   private void generate() {
      final int count = 10000000;

      // Create a GUI with the update info
      final ProgressDialog dialog = new ProgressDialog(this, "Please
                                                    wait...");

      // Create the numbers generator
      NumberGenerator generator = new NumberGenerator(count, dialog);

      dialog.setWorker(generator);

      // Add a listener triggering the update info
      generator.addPropertyChangeListener(new PropertyChangeListener() {
         public void propertyChange(PropertyChangeEvent evt) {
            if ("progress".equals(evt.getPropertyName())) {
               dialog.setValue((Integer) evt.getNewValue());
            }
         }
      });

      // Start the generation process
      generator.execute();

      // Show the modal dialog
      dialog.setVisible(true);
   }

   // [...]
```

Rozbudowa klasy SwingWorker

- Stwórzmy klasę ułatwiającą pracę z GUI oraz *SwingWorker'em*
  - Funkcjonalność do przetwarzania danych (`SwingWorker`),
  - Wyświetlanie informacji o postępach za pomocą GUI.
- Klasa `ProgressWorker<finalResult, partialResult>`
  - Dziedziczy ze `SwingWorker`,
  - Zawiera zmodyfikowaną instancję `ProgressDialog`.

Rozbudowa klasy SwingWorker (2)

- Konstruktor

```
public abstract class ProgressWorker<finalResult, partialResult> extends
SwingWorker<finalResult, partialResult> {
    private ProgressDialog progressDialog;

    /**
     * Creates a new instance presenting the user's message.
     * @param progressMessage
     */
    public ProgressWorker(String progressMessage) {
        super();

        progressDialog = new ProgressDialog(null, progressMessage);
        progressDialog.setWorker(this);

        // Add a listener to have updates about the progress
        addPropertyChangeListener(
                new PropertyChangeListener() {

                    public void propertyChange(PropertyChangeEvent evt) {
                        if ("progress".equals(evt.getPropertyName())) {
                            progressDialog.setValue((Integer) evt.getNewValue());
                        }
                    }
                }
        );
    }

    // [...]
```

Rozbudowa klasy SwingWorker (3)

- Specjalne metody
  - `Start()`
  - `done()`.

```
public abstract class ProgressWorker<finalResult, partialResult>
                        extends SwingWorker<finalResult, partialResult>
{
    private ProgressDialog progressDialog;

    // [...]

    /**
     * Starts the long-lasting activity and shows GUI.
     * This method should be executed instead of the execute() method.
     */
    public void start() {
        execute();

        progressDialog.setVisible(true);
    }

    /**
     * Executed automatically when all calcutaions are finished or
cancelled.
     * Call the super method and hides the GUI.
     */
    @Override
    protected void done() {
        super.done();

        // Hide GUI
        progressDialog.setVisible(false);
    }
```

Wykorzystanie klasy `ProgressWorker`

```
public class NumberGenerator extends ProgressWorker<List<Integer>, Void> {
    private List<Integer> numbers;
    private int maxCount;

    public NumberGenerator(int maxCount) {
        // We need to call the super-class constructor with the user's message
        super("Generating numbers (" + maxCount + ")... Please wait...");

        // System.out.println("Generowanie liczb (" + maxCount + ")... Prosze czekac");

        this.maxCount = maxCount;
    }

    @Override
    protected List<Integer> doInBackground() throws Exception {
        final int maxNumber = 1000;
        int i = 1;

        numbers = new LinkedList<Integer>();

        while(i < maxCount && isCancelled() == false) {
            Integer curInt = new Integer((int)(Math.random() * (double) maxNumber));
            numbers.add(curInt);

            if(i % 100 == 0) {
                // Update progress info
                setProgress((int)(100.0 * (float)i / maxCount));
            }

            i++;
        }

        return numbers;
    }
}
```

Wykorzystanie klasy `ProgressWorker` (2)

```
public class NumberFinder extends ProgressWorker<Integer, Void> {
    private List<Integer> numbers;
    Integer numberToFind;

    public NumberFinder(List<Integer> numbers, Integer numberToFind) {
        super("Counting occurencies of the number: " + numberToFind + "...");

        this.numberToFind = numberToFind;
        this.numbers = numbers;
    }
    @Override
    protected Integer doInBackground() throws Exception {
        int count = 0;
        int i = 0;

        for(Integer number : numbers) {
            if(isCancelled()) {
                return new Integer(count);
            }

            if(number.equals(numberToFind)) {
                count++;
            }

            if(i % 100 == 0) {
                // Update progress info
                setProgress((int)(100.0 * (float)i / numbers.size()));
            }

            i++;
        }

        return new Integer(count);
    }}
```

Wykorzystanie klasy `ProgressWorker` (3)

```
public class GeneratorAndFinder extends javax.swing.JFrame {
    // [...]

    private void GenerateFind() throws NumberFormatException, HeadlessException {

        try {
            // Generate a list with numbers
            NumberGenerator gen = new NumberGenerator((Integer)
                                                    spnCount.getValue());
            gen.start();

            // Check if cancelled
            if(gen.isCancelled()) {
                JOptionPane.showMessageDialog(null, "The operation has been
                                                        cancelled.");
                return;
            }

            // Get the result list
            List<Integer> numbers = gen.get();

            JOptionPane.showMessageDialog(null, "The generated list constains: "
                + numbers.size() + " numbers.\nPress OK to start counting
                                                    occurencies.");

             // [...]
```

Wykorzystanie klasy `ProgressWorker` (4)

```
public class GeneratorAndFinder extends javax.swing.JFrame {
    // [...]

    private void GenerateFind() throws NumberFormatException, HeadlessException {

        try {

            // cont.

            // Count the occurencies of the given number
            NumberFinder finder = new NumberFinder(numbers,
                                  Integer.parseInt(txtNumberToFind.getText()));
            finder.start();

            // Check if cancelled
            if(finder.isCancelled()) {
                JOptionPane.showMessageDialog(null, "The operation has been cancelled.");
                return;
            }
            // Get the result
            int count = finder.get();
            // Show the result
            JOptionPane.showMessageDialog(null, "The number " + txtNumberToFind.getText()
                                                + " occured " + count + " times.");
        } catch [...]
    }
```

Aplikacja MDI

- Podział kodu na pakiety, np.
  - `mt.mas.sampleapplication`
  - `mt.mas.sampleapplication.data`
  - `mt.mas.sampleapplication.gui`
- Tworzymy dedykowane okna dla poszczególnych funkcjonalności (`JInternalFrame`)
- Każdą „merytoryczną” funkcjonalność opakowujemy w oddzielną metodę.
- Wykorzystujemy `ProgressWorker` aby pokazywać postęp operacji.

Tworzenie aplikacji MDI

Demonstracja przy użyciu NetBeans

Automatyczne tworzenie diagramów

- Korzystamy z np. Net Beans (rozbudowana wersja)
  - Doinstalować plugin „UML”: menu *Tools/Plugins*.
  - Utworzyć projekt UML na podstawie istniejącego projektu Java: *File/New Project* ➔ *UML* ➔ *Reverse Engineered Java-Platform Model*
  - W projekcie Java, z menu kontekstowego zaznaczyć element (np. klasę, klasy, pakiet, metodę, itp.) i wybrać opcję *Create Diagram from Selected Element*.
  - Wybrać określony rodzaj diagramu.

Podsumowanie

- Graficzne edytory GUI bardzo ułatwiają tworzenie Graficznego Interfejsu Użytkownika.
- Aby zaimplementować GUI prawidłowo reagujące na długotrwałe przetwarzanie danych należy wykorzystać wątki.
- Pomocna przy takiej implementacji może być klasa *SwingWorker* oraz jej zmodyfikowana wersja *ProgressWorker*.
- Rozbudowane aplikacje warto tworzyć jako MDI.

Pliki źródłowe

- Pobierz pliki źródłowe do wszystkich wykładów MAS

http://www.mtrzaska.com/plik/mas/mas-source-files-lectures