Praktikum „SEP: Java-Programmierung“ SS 2019

JAVA Swing

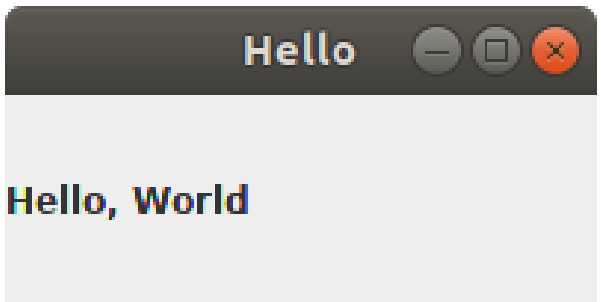
Thomas Bunk und Karlheinz Friedberger

Graphische Bedienoberflächen

- AWT
  - Abstract Windowing Toolkit
  - import `java.awt.*`;
- Swing ist eine Bibliothek von Java
  - Programmierung graphischer Benutzeroberflächen nach dem Baukastenprinzip
  - Nachfolger des AWT; setzt aber darauf auf
  - Plattformunabhängigkeit
  - Durchgängig objektorientiert
  - Eher langsam, da alle Grafikbefehle fast komplett in Java realisiert sind

Swing Grundgerüst

Jetzt endlich Code: Hallo Swing Welt!



Swing Grundgerüst

```
import javax.swing.*;

public class HelloWorld {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hello"); // 1.
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // 2.
    frame.add(new JLabel("Hello, World")); // 3.

    frame.setSize(200, 100); // 4.
    frame.setVisible(true); // 5.
  }
}
```

1. Frame erzeugen
2. Verhalten beim Schließen festlegen
   - Default-Verhalten versteckt nur das Fenster
   - `DISPOSE_ON_CLOSE` beendet auch das Programm
3. Komponenten hinzufügen
4. Größe festlegen
5. Frame anzeigen

Schaltflächen

```
import java.awt.*; javax.swing.*;

public class ButtonExample {

  public static void main(String[] args) {
    JFrame frame = new JFrame("Button example");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    Container contentPane = frame.getContentPane();

    contentPane.setLayout(new FlowLayout());
    contentPane.add(new JButton("Click me!"));
    contentPane.add(new JButton("Ignore me!"));
    contentPane.add(new JTextField("Type something, please"));

    frame.setSize(200, 150);
    frame.setVisible(true);
  }
}
```

- Aber: noch keine Reaktion auf Benutzerinteraktion

Benutzerinteraktion

- GUI-Programmierung ist *event driven programming*
- Events
  - Mausklicks
  - Mausbewegungen
  - Eingaben auf der Tastatur
  - ...
- Verschiedene Event Kategorien
  - Action
  - Keyboard
  - Mouse
  - MouseMotion
  - ...
- Wie bekommt man Events mit
  - Registrieren eines *Event-Listeners* bei der jeweiligen Swing Komponente

Beispielcode Benutzerinteraktion (1/2)

```
public class ActionListenerDemo {

  public static void main(String[] args) {
    JFrame frame = new JFrame("Button example");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    JButton button1 = new JButton("Click me!");
    JButton button2 = new JButton("Ignore me!");
    JTextField textField = new JTextField("Type something here");

    button1.addActionListener(new ActionListener() {

      @Override
      public void actionPerformed(ActionEvent e) {
        textField.setText("First button");
      }
    });
```

Beispielcode Benutzerinteraktion (2/2)

```
    button2.addActionListener(new ActionListener() {

      @Override
      public void actionPerformed(ActionEvent e) {
        textField.setText("Second button");
      }
    });

    Container contentPane = frame.getContentPane();
    contentPane.setLayout(new FlowLayout());
    contentPane.add(button1);
    contentPane.add(button2);
    contentPane.add(textField);

    frame.setSize(200, 150);
    frame.setVisible(true);
  }
}
```

Event Processing

Ablauf der Ereignisbehandlung

1. Benutzer interagiert mit Swing-Komponente
2. Swing erzeugt Event für die Interaktion und Komponente
3. Swing benachrichtigt Event Listener der Komponente
4. Event Listener-Implementierungen stoßen die eigene Programmlogik an

Auf Ereignisse reagieren

- `ActionListener` können u.a. registriert werden auf:
  - JButton / JToggleButton (Click)
  - JTextField (Enter)
  - JComboBox (Selektion)
  - JCheckBox (Selektion)
  - JMenuItem (Click)

- `ChangeListener` werden benachrichtigt, wenn sich der Zustand einer Komponente ändert, z.B.
  - JSlider, JSpinner, JProgressBar, ...

- `ComponentListener` können auf jeder Komponente registriert werden
  - Benachrichtigungen: resize, move, shown, hidden, ...

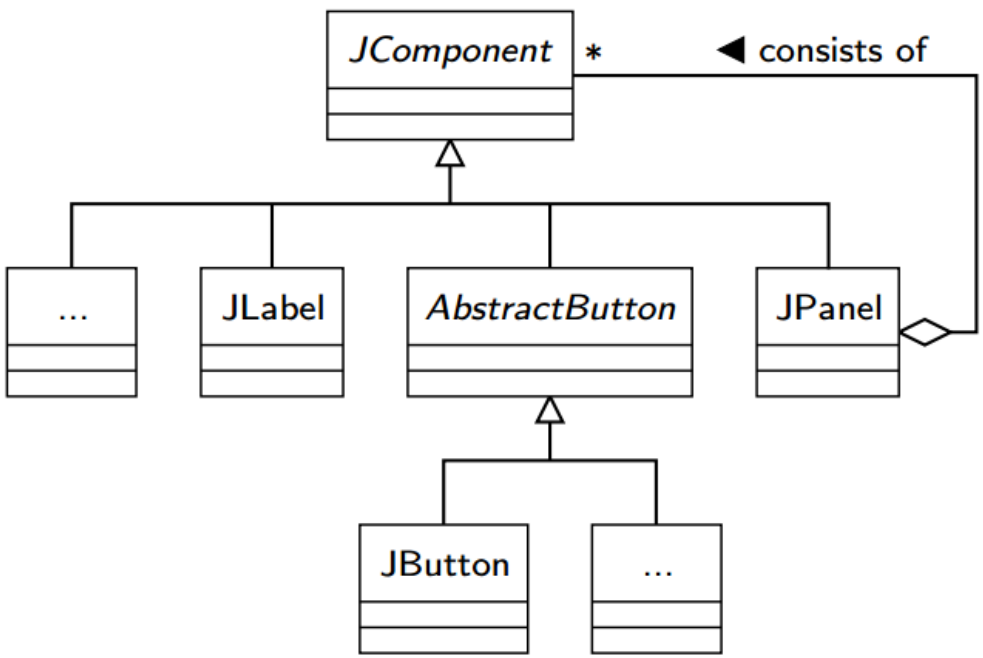
EventListener vs. EventAdapter

- Was tun, wenn nur eine Methode (von vier) interessant ist?
- ... z.B. `mouse released`

- Hierzu gibt es Adapter
  - Abstrakte Klasse, die alle Methoden leer implementiert
  - Benötigte Methoden können überschrieben werden

Hierarchischer Aufbau von GUIs

- Basis Elemente sind elementare GUI Komponenten
  - `JButton`, `JTextField`, `JLabel`, ...
- Container sind Behälter für andere Komponenten
  - `JFrame`, `JPanel`, ...
- In UML



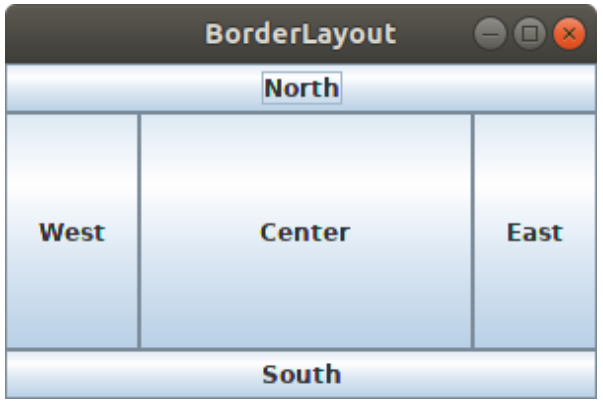
Layout Manager

Layout Manager ordnen Elemente in Containern an

- `FlowLayout`
  - Anordnung nach Einfügereihenfolge
  - Von links nach rechts, von oben nach unten
- `BorderLayout`
  - Einfache Positionsangabe
  - Oben, unten, links, rechts mittig
- `GridLayout`
  - Tabellenlayout mit fester Zeilen- und Spaltenzahl
  - Alle Zellen gleich groß
- `GridBagLayout`
  - Richtet Komponenten horizontal und vertikal aus
  - Verschieden große Komponenten möglich
- ...

Beispiel: BorderLayout

```
public class BorderLayoutExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("BorderLayout");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    // forwarder to setLayout(...) of content pane
    frame.setLayout(new BorderLayout());

    Container c = frame.getContentPane();
    c.add(new JButton("North"), BorderLayout.NORTH);
    c.add(new JButton("South"), BorderLayout.SOUTH);
    c.add(new JButton("East"), BorderLayout.EAST);
    c.add(new JButton("West"), BorderLayout.WEST);
    c.add(new JButton("Center"), BorderLayout.CENTER);

    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```



Beispiel: GridLayout

```
public class GridLayoutExample {

  public static void main(String[] args) {
    JFrame frame = new JFrame("GridLayout");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    frame.setLayout(new GridLayout(4, 5));

    Container c = frame.getContentPane();
    for (int i = 0; i < 18; i++) {
      c.add(new JButton(String.valueOf(i)));
    }

    frame.setSize(300, 300);
    frame.setVisible(true);
  }
}
```

Hierarchisches GUI Design

```
public class HierarchicLayoutExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hierarchic layout");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    frame.setLayout(new BorderLayout());

    JPanel c = new JPanel(new GridLayout(2, 3));
    for (int i = 0; i < 6; ++i) {
      c.add(new JButton(String.valueOf(i)));
    }
    frame.getContentPane().add(c, BorderLayout.CENTER);

    JPanel w = new JPanel();
    w.setLayout(new BoxLayout(w, BoxLayout.Y_AXIS));
    for (int i = 0; i < 5; ++i) {
      w.add(Box.createVerticalGlue());
      w.add(new JLabel(String.valueOf(i)));
    }
    w.add(Box.createVerticalGlue());
    frame.getContentPane().add(w, BorderLayout.WEST);

    frame.pack();
    frame.setVisible(true);
  }
}
```

Zusammenfassung GUI-Aufbau

- GUI-Aufbau mit Swing
  - Aus verschiedenen Komponenten
  - Verschachtelte Container für hierarchischen Aufbau
  - Layout-Manager zur Anordnung der Komponenten

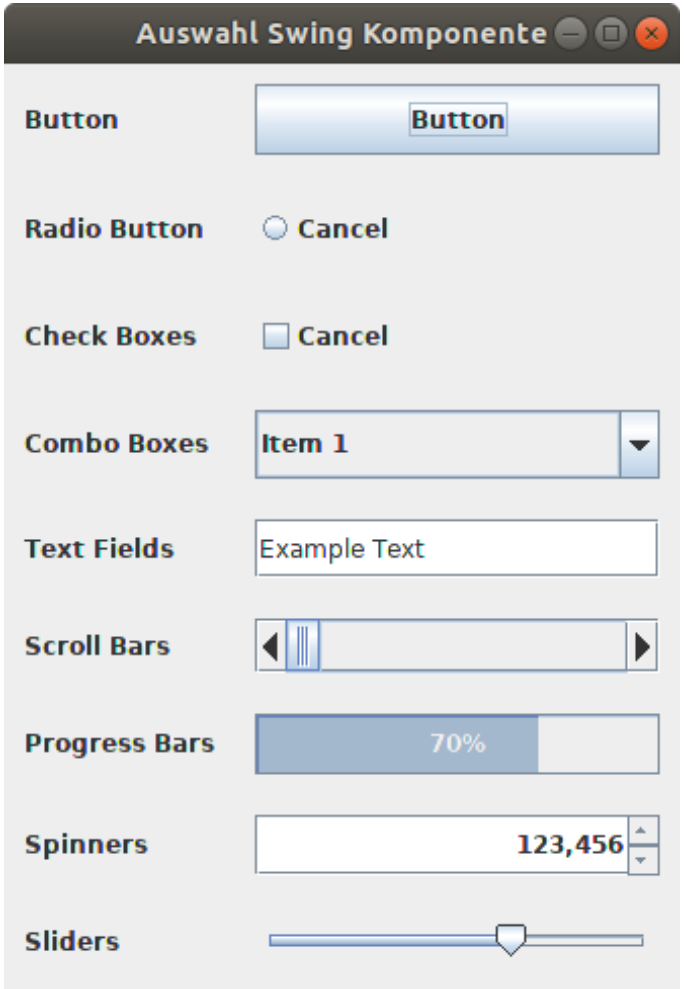
- Setzen der Container mit `setLayout(new FlowLayout())`

- Mehr dazu: https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

Mehr Schaltflächen

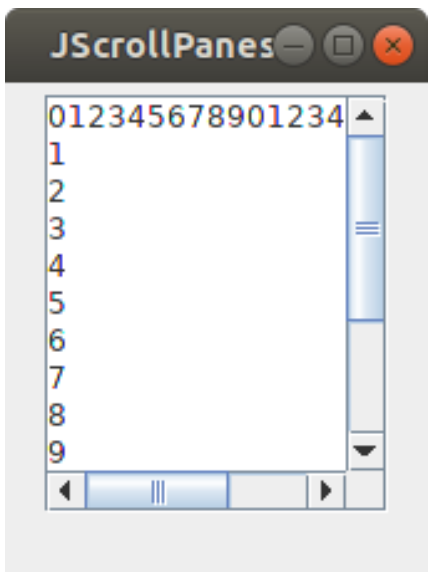
Auswahl an Komponenten für verschiedene Zwecke

- `JButton`
- `JRadioButton`
- `JCheckBox`
- `JComboBox`
- `JTextField`
- `JScrollBar`
- `JProgressBar`
- `JSpinner`
- `JSlider`
- ...



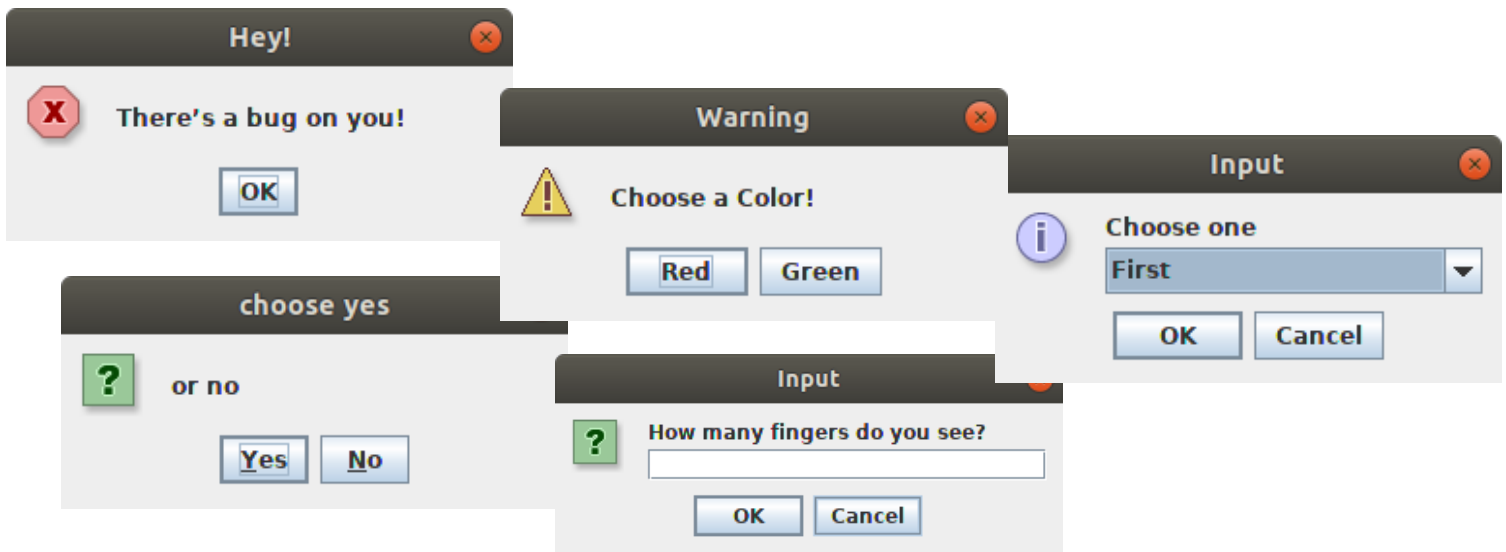
Scrollen

```
public class ScrollPaneExample {

  public static void main(String[] args) {
    JFrame frame = new JFrame("JScrollPanes");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    Container cp = frame.getContentPane();
    cp.setLayout(new FlowLayout());

    String content = "012345678901234567890123456789012" +
        "\n1\n2\n3\n4\n5\n6\n7\n8\n9\n0\n1\n2\n3\n4\n5";
    JTextArea ta = new JTextArea(content, 10, 10);
    cp.add(new JScrollPane(ta,
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS));

    frame.setSize(170, 230);
    frame.setVisible(true);
  }
}
```



Einfache Popup-Dialoge (1/2)

- Neben Basiskomponenten gibt es auch vorgefertigte Standarddialoge
- `JOptionPane` – bieten viele konfigurierbare Dialoge an:
  - `showMessageDialog`: Warnungen / Informationen ausgeben
  - `showConfirmDialog`: Bestätigung erfragen
  - `showInputDialog`: Zur Eingabe auffordern



Einfache Popup-Dialoge (2/2)

- Reiner Hinweis:

```
JOptionPane.showMessageDialog(null, "There's a bug on you!",
                "Hey!", JOptionPane.ERROR_MESSAGE);
```

- Auswahldialog (ja/nein, links/rechts, ...):

```
JOptionPane.showConfirmDialog(null, "or no", "choose yes",
                JOptionPane.YES_NO_OPTION);
```

- Eingabedialog:

```
String val = JOptionPane.showInputDialog(
                "How many fingers do you see?");
```

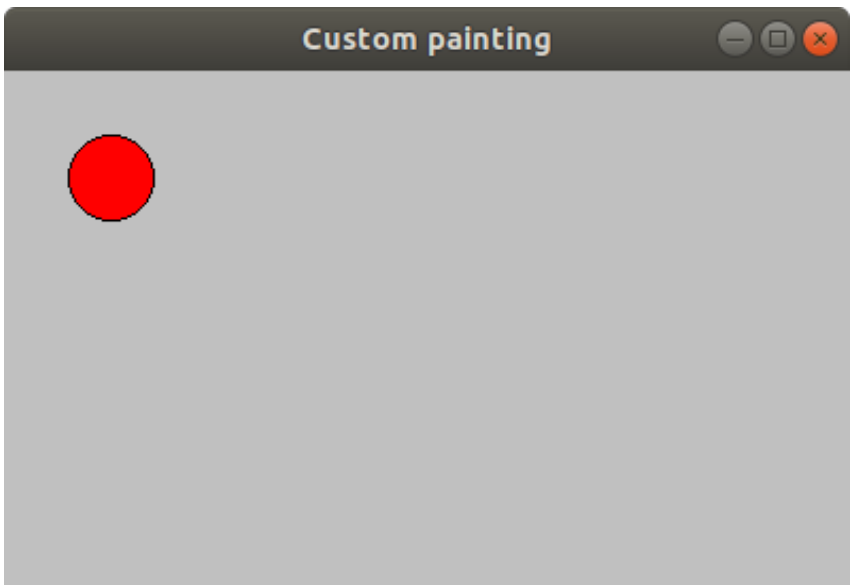
- Frei konfigurierbarer Dialog:

```
String[] options = {"Red", "Green"};
int sel = JOptionPane.showOptionDialog(null,
    "Choose a Color!", "Warning", JOptionPane.DEFAULT_OPTION,
    JOptionPane.WARNING_MESSAGE, null, options, options[0]);
```

  - Parameter: Elternkomponente, Nachricht, Titel, Optionsart (ja/nein, ...), Nachrichtentyp, Icon, Optionen, Initialwert
- Alle `JOptionPane` Dialoge sind *modal*
  - Aufrufer / aufrufendes Fenster wird blockiert bis Dialog beendet ist

Zeichnen mit Swing

- Manchmal reichen Standard-Komponenten nicht aus
  - Selbst zeichnen notwendig
- `JPanel` als Zeichenfläche
- Beispiel: Roter Kreis mit schwarzem Rand auf Mausposition



Grundgerüst

1. JPanel Subklasse erzeugen
2. JPanel `paintComponent(Graphics g)` überschreiben
3. Graphics bietet Methoden zum Zeichnen an
4. Neuzeichnen mit `repaint()` anfordern

- Interessante Methoden des Graphic-Objekts
  - `setColor`, `setFont`
  - `drawLine`, `drawOval`, `drawRect`, (...)
  - `fillOval`, `fillRect`, (...)
  - ...

Vollständiges Beispiel (1/3)

```
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class CustomPaintingFrame extends JFrame {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        @Override
        public void run() {
          showFrame();
        }
    });
  }

  private static void showFrame() {
    new CustomPaintingFrame().setVisible(true);
  }

  public CustomPaintingFrame() {
    setTitle("Custom painting");
    setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    createContent();
  }

  private void createContent() {
    CustomPaintingPanel panel = new CustomPaintingPanel();
    setContentPane(panel);
    setSize(400, 300);
  }
}
```

Vollständiges Beispiel (2/3)

```
import java.awt.*;
import javax.swing.JPanel;

public class CustomPaintingPanel extends JPanel {

  private static final int RADIUS = 20;
  private static final int DIAMETER = 2 * RADIUS;
  private Point position;

  public CustomPaintingPanel() {
    position = new Point(50, 50);
    addEventListener();
  }

  private void addEventListener() {
    addMouseListener(new MouseAdapter() {

      @Override
      public void mouseClicked(MouseEvent e) {
        updatePosition(e);
      }
    });
  }

  private void updatePosition(MouseEvent e) {
    position.setLocation(e.getPoint());
    repaint();
  }
```

Vollständiges Beispiel (3/3)

```
  @Override
  protected void paintComponent(Graphics g) {
    g.setColor(Color.LIGHT_GRAY);
    g.fillRect(0, 0, getWidth(), getHeight());

    int xPos = position.x - RADIUS;
    int yPos = position.y - RADIUS;

    // Kreis ausfuellen
    g.setColor(Color.RED);
    g.fillOval(xPos, yPos, DIAMETER, DIAMETER);

    // Umrisslinie zeichnen
    g.setColor(Color.BLACK);
    g.drawOval(xPos, yPos, DIAMETER, DIAMETER);
  }
}
```

Hinweise zum Zeichnen mit Swing

- Meistens rechnen relativ zur Größe des Panels notwendig
- Koordinatenursprung links oben
  - y-Koordinaten steigen von oben nach unten
- Nur geänderte Komponenten neuzeichnen lassen, nicht ganze GUI
  - Hier nur das JPanel und nicht das ganze JFrame
- Für fertige GUI Komponenten aus der API `repaint()` meist nicht notwendig bzw. überflüssig
  - Zustandsverändernde Methoden rufen `this.repaint()` selbst auf, z.B. Methode `setText(...)` von JLabel- oder JTextField-Objekten

Trennung von Model und View

- Grundprinzip des Software Engineerings
  - Jede Klasse hat genau ein Geheimnis!
- Problem: Verflechtung von Programmlogik und Darstellung
  - Algorithmik (Berechnung von Daten) / Business Logik in der Event-Verarbeitung
  - Event-Verarbeitung in den graphischen Komponenten
- Konsequenz
  - Schwer wartbar
  - Schwer erweiterbar
  - Nicht wiederverwendbar
- Lösung
  - Konsequente Trennung

Beispielcode zur Trennung von Model und View (1/3)

```
public class BusinessLogic {
  private int modifier;

  public BusinessLogic(int modifier) {
    this.modifier = modifier;
  }

  public void setModifier(int modifier) {
    this.modifier = modifier;
  }

  public int getModifier() {
    return modifier;
  }

  // some business operations

  public int calculation1(int arg) {
    return arg * modifier;
  }

  public int calculation2(int arg) {
    return arg + modifier;
  }
}
```

Beispielcode zur Trennung von Model und View (2/3)

```
public class View extends JFrame {

  private JTextField tf = new JTextField(15);
  private JButton calc1 = new JButton("Calc 1");
  private JButton calc2 = new JButton("Calc 2");
  private BusinessLogic bl = new BusinessLogic(2);

  public static int getValue(JTextField textField) {
    try {
      return Integer.parseInt(textField.getText());
    } catch (NumberFormatException e) {
      return 0;
    }
  }

  class Calc1Listener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent event) {
      // delegation to BusinessLogic
      int i = bl.calculation1(getValue(tf));
      tf.setText(Integer.toString(i));
    }
  }
```

Beispielcode zur Trennung von Model und View (3/3)

```
  class Calc2Listener implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
      // delegation to BusinessLogic
      int i = bl.calculation2(getValue(tf));
      tf.setText(Integer.toString(i));
    }
  }

  public View() {
    setupFrame();
    createContent();
    addEventListeners();
  }

  ...
}
```

Trennung von Model und View

Vorteile dieser Architektur:

- Unabhängigkeit der Geschäftslogik von der GUI
  - Wiederverwendbarkeit
  - Automatisch testbar
- Delegation der Berechnungen an die Geschäftslogik
- Änderungen an Model und View jeweils unabhängig vom anderen möglich

GUI Programmierung und Nebenläufigkeit

- *Event-driven programming*
  - Ablauf des Programms wird bestimmt durch Ereignisse wie Nutzerinteraktionen (Mausklicks, Tastatureingaben), I/O-Ereignisse, usw.

- Anforderung an Anwendungen mit graphischer Oberfläche ist *reaktives* Verhalten:
  - GUI muss jederzeit auf Eingaben des Benutzers reagieren können

- Deshalb: Anwendungen mit Benutzer-Interaktion sind immer nebenläufig
  - Steuerung der Oberfläche wird durch einen eigenen UI-Thread abgewickelt
  - Innerhalb dieses Threads werden Aktionen im Zusammenhang mit der GUI sequentiell abgearbeitet

Event Dispatch Thread (EDT)

Der UI-Thread in Swing: *Event dispatch thread* (EDT)

- nimmt alle Ereignisse an der GUI (z.B. Klicks) an und liefert sie an den zuständigen Empfänger aus
- nimmt alle Änderungen an den GUI-Elementen vor

Swing Thread Design

- Swing ist (in weiten Teilen) nicht thread-safe!
- Zwei Grundregeln, um fehlerfrei funktionierendes Programm zu gewährleisten:

1. Jede GUI Aktivität findet im EDT statt!
   - Beispiel: Erzeugung und Zugriff auf GUI Objekte

2. Keine weiteren (zeitaufwändigen) Aktivitäten in diesem Thread!
   - Blockierende Aktionen (z.B. I/O) absolutes No-Go
   - Ist der UI-Thread mit langwierigen Operationen beschäftigt, dann hängt die Anwendung

GUI Aktivitäten im EDT

- Niemals Swing Aufrufe (einschließlich Konstruktor-Aufrufe) aus einem anderem Thread als den EDT vornehmen
- Aufrufe von Swing Methoden außerhalb des EDTs wie folgt:
  - `SwingUtilities.invokeLater(Runnable)` // Asynchron
  - `SwingUtilities.invokeAndWait(Runnable)` // Synchron
- Weitere Ausnahmen (threadsichere Methoden):
  - `repaint()`
  - `revalidate()`
  - `add...Listener()`, `remove...Listener()`

Beispiel Hello World

```
public class HelloWorld {
  private static void createAndShowGUI() {
    JFrame frame = new JFrame("Hello World");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    JLabel label = new JLabel("Hello World");
    frame.getContentPane().add(label);
    frame.pack();
    frame.setVisible(true);
  }

  public static void main(String[] args) {
    // schedule a job for the event-dispatching thread:
    // creating and showing this application's GUI
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      @Override
      public void run() {
        createAndShowGUI();
      }
    });
  }
}
```

- Für alternatives Beispiel, s. Folien 24 – 26

Reaktivität garantieren: Hängende GUI vermeiden

- Ist UI-Thread mit langwierigen Operationen beschäftigt, dann hängt die UI
- Beispiel:

```
public class FreezingFrame extends JFrame {
  private JButton freezeButton;
  private JLabel status;

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {

      @Override
      public void run() {
        new FreezingFrame().setVisible(true);
      }
    });
  }

  public FreezingFrame() {
    setTitle("Freezing Frame");
    setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    createContent();
    createEventListeners();
  }
```

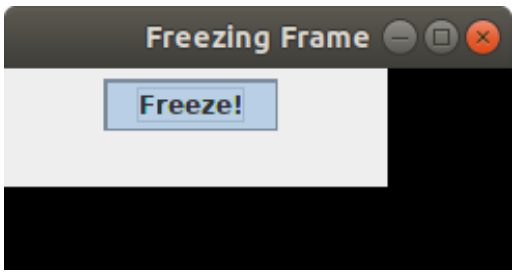
Beispiel (Fortsetzung)

```
  private void createContent() {
    freezeButton = new JButton("Freeze!");
    status = new JLabel();

    JPanel panel = new JPanel();
    setContentPane(panel);
    panel.add(freezeButton);
    panel.add(status);

    setBounds(50, 50, 200, 100);
  }

  private void createEventListeners() {
    freezeButton.addActionListener(new ActionListener() {

      @Override
      public void actionPerformed(ActionEvent e) {
        freeze();
      }
    });
  }

  private void freeze() {
    try {
      status.setText("Freezing frame...");
      Thread.sleep(10000); // simulate some hard work
      status.setText("Done.");
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
  }
}
```

Zeitaufwändige Operationen im EDT (1/2)

- Aufwendige Aktionen im EventDispatcher machen GUI träge

- Problem im Code von vorherigen zwei Folien:
  1. Die GUI wird normal angezeigt
  2. Bei Klick auf Button "friert" GUI ein
  3. `actionPerformed` hindert EventDispatcher am Zeichnen



- Grund: Beschäftigter EDT kann keine Events mehr abarbeiten

Zeitaufwändige Operationen im EDT (2/2)

- Lösung:
  - Zeitintensive Operationen in eigene (Worker-)Threads auslagern und im Hintergrund ausführen lassen
  - Im Event Dispatcher nur reine GUI-Updates, sodass dieser weiter Benutzereingaben annehmen kann
  - Worker-Thread sollte niemals direkt in die GUI eingreifen, sondern stattdessen den EDT mit Änderungen an der GUI beauftragen
- Grundprinzip: GUI darf nie einfrieren
- `repaint()` ist nur ein Hinweis für den EventDispatcher
  - Zeichnet erst, wenn er Zeit hat

SwingWorker

Swing stellt hierfür die Klasse `SwingWorker<T,V>` bereit

- `doInBackground()` läuft im Hintergrund und führt die eigentliche Berechnung durch
- `T` ist der Typ des Endergebnisses (d.h. der Rückgabewert von `doInBackground()`)

Optionale Methoden:

- `done()` läuft im EDT
  - Verwenden, um Ergebnis an die GUI zu senden
- `V` ist der Typ von Zwischenergebnissen, und können einen anderen Typ als das Endergebnis besitzen
  - `publish()/process(V)` verwenden, um Zwischenergebnisse an die GUI zu senden

Lösung zu vorherigem Beispiel

- `freeze()`-Methode aus vorherigem Beispiel durch folgende `doNotFreeze()`-Methode ersetzen:

```
private void doNotFreeze() {
  SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>() {

    @Override
    protected Void doInBackground() throws Exception {

      SwingUtilities.invokeLater(new Runnable() {
        @Override
        public void run() {
          // Update the text of JLabel before starting the hard work.

          // However, as the doInBackground() runs outside of the EDT,
          // this is invoked within the SwingUtilities#invokeLater(...)-method
          status.setText("Freezing frame...");
        }
      });

      Thread.sleep(10000); // simulate some hard work
      return null;
    }

    @Override
    protected void done() {
      status.setText("Done.");
    }
  };
  worker.execute();
}
```