

9. GUI Applications

vorgetragen durch
Maraike Stuffer
am 20.6.2008

Gliederung

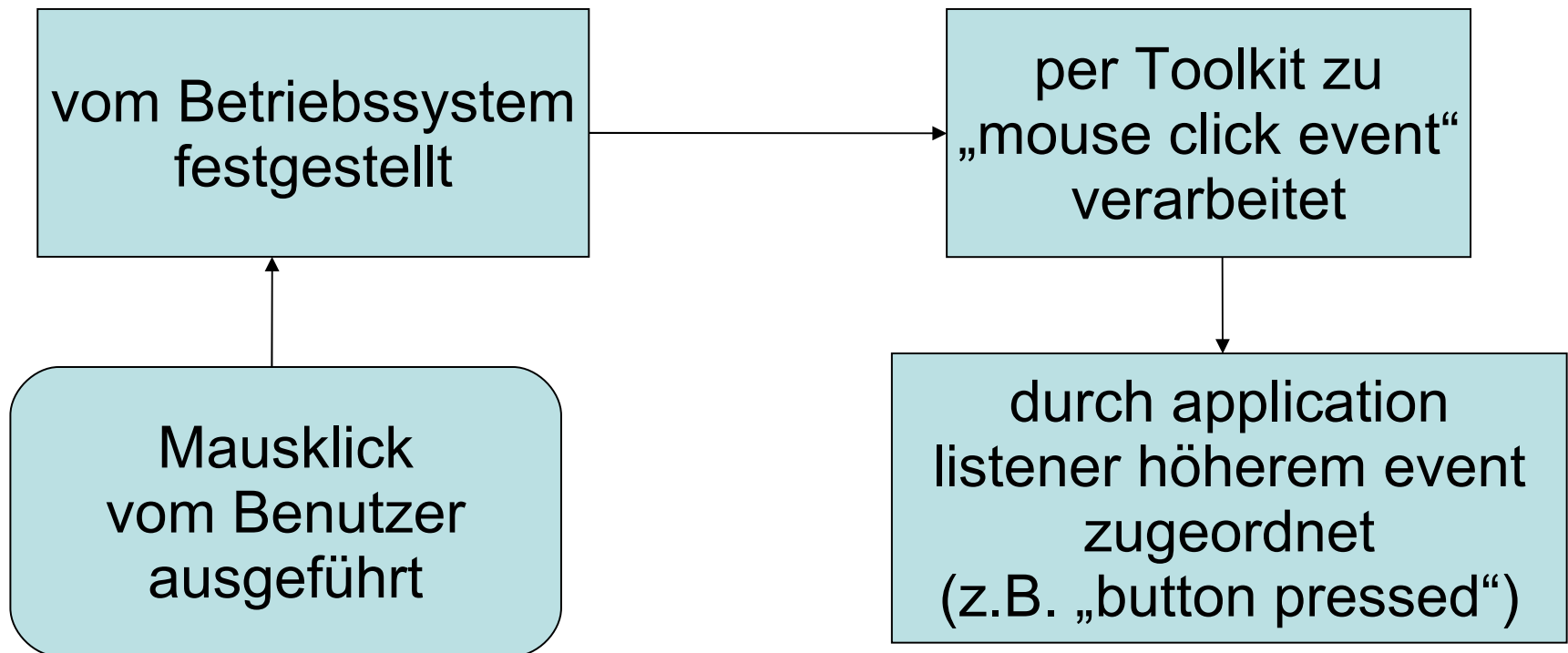
- 9.1 Warum sind GUIs single-threaded?
 - 9.1.1 Sequential event processing
 - 9.1.2 Thread confinement in Swing
- 9.2 Short-running GUI tasks
- 9.3 Long-running GUI tasks
 - 9.3.1 Cancellation
 - 9.3.2 Progress and completion indication
 - 9.3.3 SwingWorker
 - Beispiel: ProgressBarDemo
- 9.4 Shared data models
 - 9.4.1 Thread-safe data models
 - 9.4.2 Split data models
- 9.5 Andere Arten von single-threaded subsystems

9.1 Warum sind GUIs single-threaded?

- Tradition
 - GUI-Anwendungen waren von Anfang an single-threaded
 - GUI events wurden von einer „main event loop“ bearbeitet, heute ähnliches System: „event dispatch thread“

9.1 Warum sind GUIs single-threaded?

- Unüberwindbare Hürden in der Entwicklung
Beispiel: Button reagiert auf Mausklick



9.1 Warum sind GUIs single-threaded?

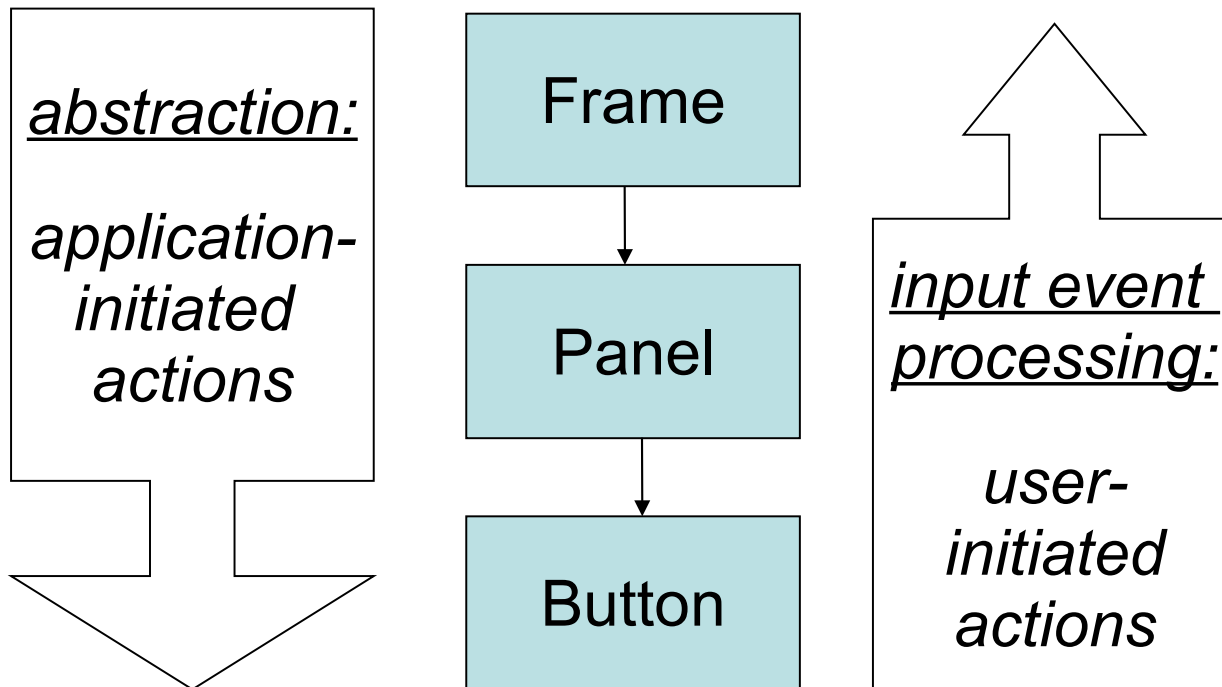
Hier kommt man (multi-threaded) in Teufelsküche:

- ❖ Activities greifen in umgekehrter Reihenfolge auf GUI-Objekte zu
- +
- ❖ Jedes Objekt soll thread-safe sein
- =
- Inconsistent lock ordering

Inconsistent lock ordering führt zu deadlock!

9.1 Warum sind GUIs single-threaded?

Beispiel für Hierarchie:



9.1 Warum sind GUIs single-threaded?

```
public static Object cacheLock = new Object();  
public static Object tableLock = new Object();
```

...

```
public void oneMethod() {  
    synchronized (cacheLock) {  
        synchronized (tableLock) {  
            doSomething();  
        }  
    }  
}
```

```
public void anotherMethod() {  
    synchronized (tableLock) {  
        synchronized (cacheLock) {  
            doSomethingElse();  
        }  
    }  
}
```

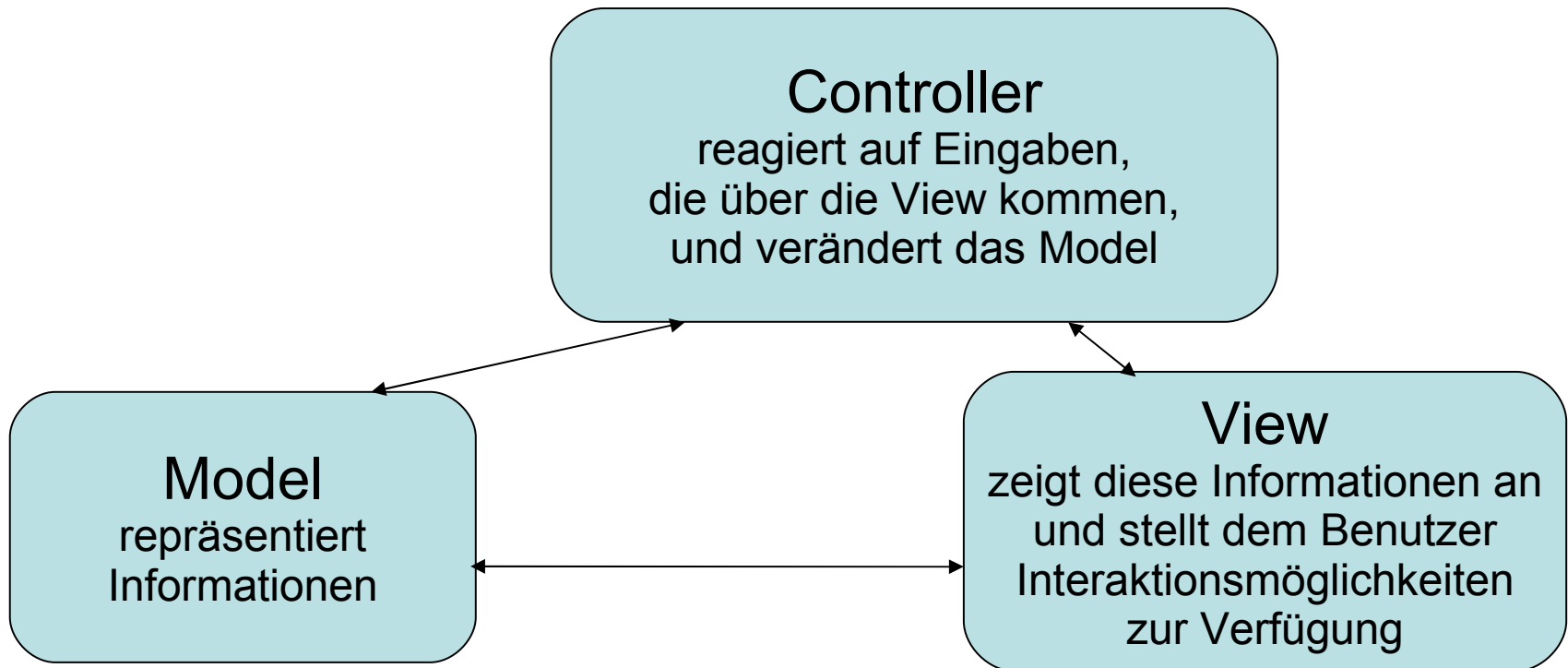


**Inconsistent lock ordering
führt zu deadlock!**

9.1 Warum sind GUIs single-threaded?

Weiterer, deadlock-begünstigender Faktor:

Hohe Verbreitung des „model-view-control pattern“

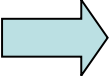
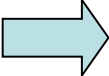


Vereinfacht die Implementierung von GUI-Anwendungen, erhöht aber das Risiko von inconsistent lock ordering

9.1 ...und wie erreichen sie überhaupt Thread-Safety?

- Durch Thread-Confinement
- Auf alle GUI-Objekte wird nur durch den event thread zugegriffen

9.1 ...und wie erreichen sie überhaupt Thread-Safety?

Multi-threaded GUI toolkits	Single-threaded GUI toolkits
<p>Man muss genau beachten, was, wie, womit zusammen hängt</p> <p>Bei sehr hohem Grad an Verständnis für das toolkit theoretisch möglich</p> <p> In der Praxis unhaltbar</p>	<p>Man muss speziell „nur“ darauf achten, dass alle Objekte korrekt confined sind</p> <p> Für den Durchschnittsprogrammierer sehr gut geeignet</p>

9.1.1 Sequential event processing

- Es existiert nur ein thread, der GUI tasks bearbeitet
- Tasks werden nacheinander abgearbeitet
- **Vorteil:**
Einfacher zu programmieren

9.1.1 Sequential event processing

- Nachteil?

9.1.1 Sequential event processing

- **Nachteil:**

Responsibility ist beeinträchtigt

9.1.1 Sequential event processing

Problem:

- Ein langandauernder Task blockiert alle anderen Tasks
- Falls diese wartenden Tasks Benutzereingaben verarbeiten, oder visuelles Feedback geben, scheint die Anwendung eingefroren zu sein (nicht einmal „Cancel“ ist verfügbar)

9.1.1 Sequential event processing

Lösung?

9.1.1 Sequential event processing

Lösung:

- Der langandauernde Task muss in einem anderen Thread ausgeführt werden, so dass der Event Thread frei wird
- Visuelles Feedback (z.B. Fortschrittsbalken) realisiert man wieder über den Event Thread

9.1.2 Thread confinement in Swing

- Alle Swing Komponenten sind confined/beschränkt auf den Event Thread
- Jeglicher Code, der auf die Objekte zugreift, muss im Event Thread ablaufen

**Swing Komponenten und Models
sollten nur vom
event-dispatching thread
erzeugt, verändert und abgerufen werden**

9.1.2 Thread confinement in Swing

Ausnahmen:

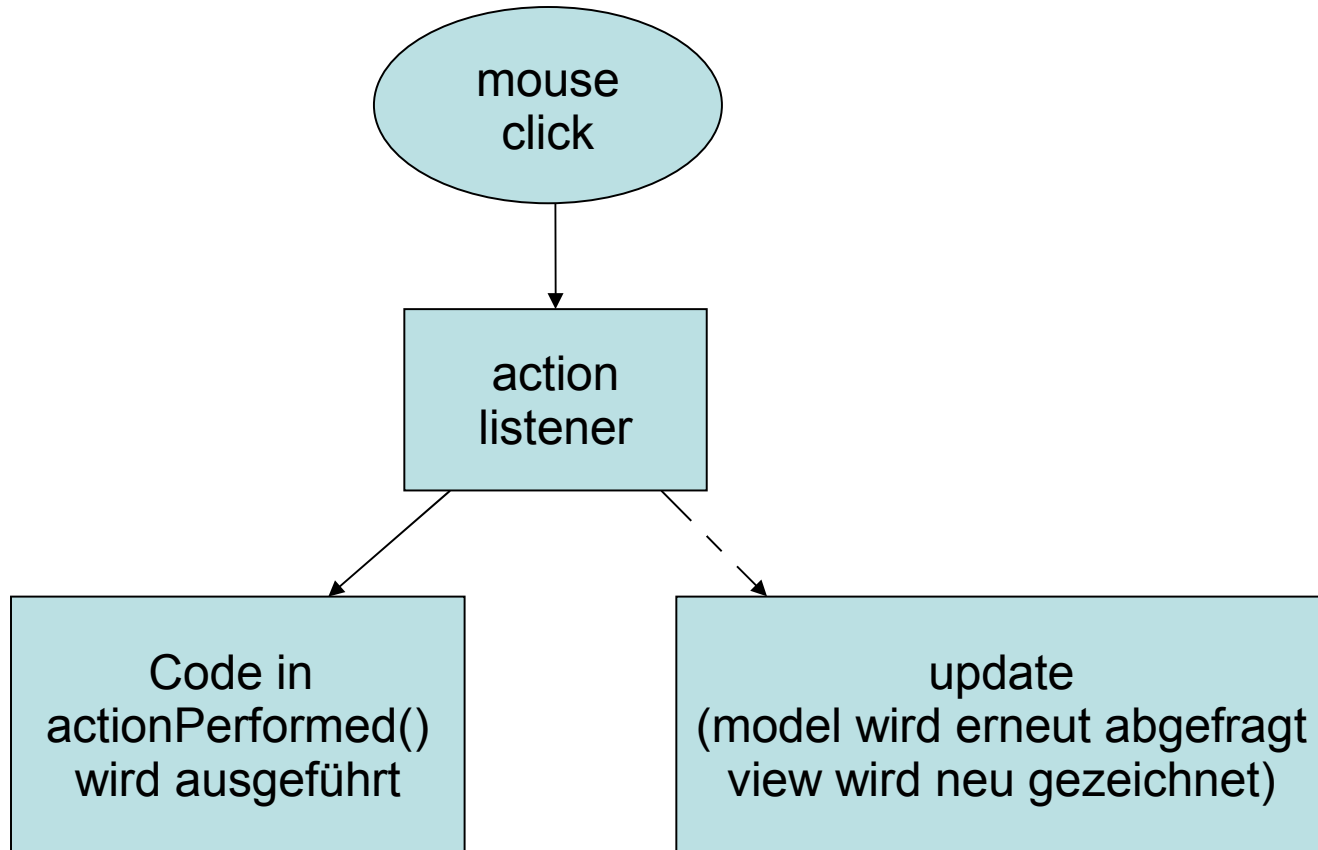
- In der Javadoc eindeutig als thread-safe bezeichnet
- `SwingUtilities.isEventDispatchThread`
- `SwingUtilities.invokeLater`
- `SwingUtilities.invokeAndWait`
- Methoden, um ein `Repaint` oder eine Gültigkeitsabfrage an die Eventqueue anzuhängen
- Methoden, um Listener hinzuzufügen oder zu entfernen

9.2 Short-running GUI tasks

- Kurze Tasks können im Event Thread ausgeführt werden
- Solange Tasks kurzlebig sind und nur auf GUI Objekte zugreifen, muss man sich so gut wie keine Sorgen über Thread-Sicherheit machen

9.2 Short-running GUI tasks

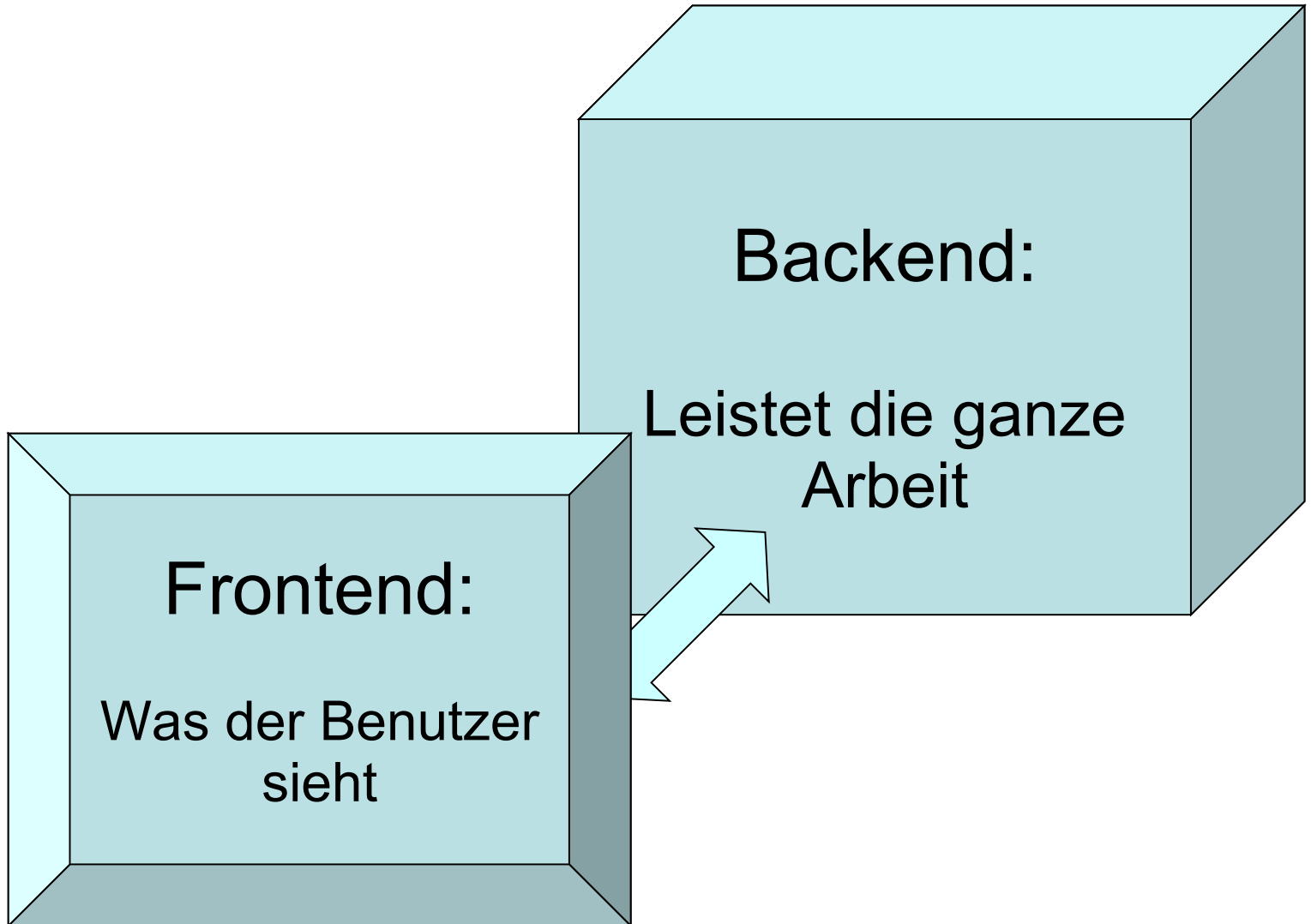
ActionListener in Model und View:



9.3 Long-running GUI tasks

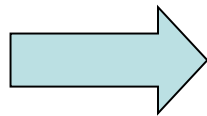
- Tasks dauern eventuell länger, als der Benutzer warten will
(z.B. Rechtschreibprüfung, Downloaden)
- Solche Tasks müssen in einem anderen Thread bearbeitet werden, damit das GUI weiterhin reagieren kann

9.3 Long-running GUI tasks



9.3 Long-running GUI tasks

- Mit Swing ist es einfach, einen Task im Event Thread laufen zu lassen, aber
- es gibt keine Mechanismen um einen Task in einem anderen Thread auszuführen



Realisierung durch Executor,
bzw einen cached thread pool

9.3 Long-running GUI tasks

Einen long-running task einem ActionListener zuordnen:

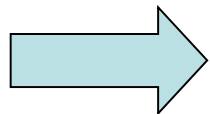
```
ExecutorService backgroundExec =  
    Executors.newCachedThreadPool();  
  
...  
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        backgroundExec.execute(new Runnable() {  
            public void run() {doBigComputation();}  
        });  
    });  
});
```

Mit User Feedback:

```
...
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button.setEnabled(false);
        label.setText(„busy“);
        backgroundExec.execute(new Runnable() {
            public void run() {
                try{doBigComputation();
            } finally {
                GuiExecutor.instance().execute(new Runnable() {
                    public void run() {
                        button.setEnabled(true);
                        label.setText(„idle“);
                    }
                });
            }
        });
    }
});
}});
```

9.3.1 Cancellation

- Benutzer möchte lange Tasks abbrechen können
- Einfache Möglichkeit: `Future`
- `mayInterruptIfRunning` auf **true** setzen



mit `Future.cancel()`

```

Future<?> runningTask = null; //thread-confined
...
startButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (runningTask != null) {
            runningTask = backgroundExec.submit(new
                                                    Runnable() {
                public void run() {
                    while (moreWork()) {
                        if (Thread.currentThread()
                            isInterrupted()) {
                            cleanUpPartialWork();
                            break:
                        } doSomeWork();
                    }
                }
            });
        }
    }
});
});

```

>>

```
cancelButton.addActionListener(new  
    ActionListener() {  
    public void actionPerformed(Action  
        Listener() {  
        if (running Task != null)  
            runningTask.cancel(true);  
    }});
```

9.3.2 Progress and completion indication

- `FutureTask` hat einen `done` hook, der completion notification vereinfacht
- `done` wird aufgerufen, wenn der background task fertig ist

9.3.3 SwingWorker

- Kombination aus `FutureTask` und `Executor` für long-running Tasks, die in background threads laufen
- Das gibt es schon: `SwingWorker` (einschließlich cancellation, completion notification und progress indication)

Beispiel: ProgressBarDemo

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.beans.*;
import java.util.Random;

public class ProgressBarDemo extends JPanel
    implements ActionListener,
        PropertyChangeListener {

    private JProgressBar progressBar;
    private JButton startButton;
    private Task task;

    class Task extends SwingWorker<Void, Void> {
        /*
         * Main task. Executed in background thread.
         */
        @Override
        public Void doInBackground() {
            Random random = new Random();
            int progress = 0;
            //Initialize progress property.
            setProgress(0);
            while (progress < 100) {
                //Sleep for up to one second.
                try {
                    Thread.sleep(random.nextInt(1000));
                } catch (InterruptedException ignore) {}
                //Make random progress.
                progress += random.nextInt(10);
                setProgress(Math.min(progress, 100));
            }
            return null;
        }
    }
}
```

```
/*
 * Executed in event dispatching thread
 */
@Override
public void done() {
    Toolkit.getDefaultToolkit().beep();
    startButton.setEnabled(true);
    setCursor(null); //turn off the wait cursor
    // taskOutput.append("Done!\n");
}
}

public ProgressBarDemo() {
    super(new BorderLayout());

    //Create the demo's UI.
    startButton = new JButton("Start");
    startButton.setActionCommand("start");
    startButton.addActionListener(this);

    progressBar = new JProgressBar(0, 100);
    progressBar.setValue(0);
    progressBar.setStringPainted(true);

    JPanel panel = new JPanel();
    panel.add(startButton);
    panel.add(progressBar);

    add(panel, BorderLayout.PAGE_START);
    setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));
}
```

```
/**
 * Invoked when the user presses the start button.
 */
public void actionPerformed(ActionEvent evt) {
    startButton.setEnabled(false);
    setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    //Instances of javax.swing.SwingWorker are not reusable, so
    //we create new instances as needed.
    task = new Task();
    task.addPropertyChangeListener(this);
    task.execute();
}
```

```
/**
 * Invoked when task's progress property changes.
 */
public void propertyChange(PropertyChangeEvent evt) {
    if ("progress" == evt.getPropertyName()) {
        int progress = (Integer) evt.getNewValue();
        progressBar.setValue(progress);
    }
}
```

```

/**
 * Create the GUI and show it. As with all GUI code, this must run
 * on the event-dispatching thread.
 */
private static void createAndShowGUI() {
    //Create and set up the window.
    JFrame frame = new JFrame("ProgressBarDemo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Create and set up the content pane.
    JComponent newContentPane = new ProgressBarDemo();
    newContentPane.setOpaque(true); //content panes must be opaque
    frame.setContentPane(newContentPane);

    //Display the window.
    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() {
        public void run() {
            createAndShowGUI();
        }
    });
}
}

```

9.4 Shared data models

- In simplen GUI-Programmen ist der einzige Thread außer des Event Threads der Main Thread
- Single-Thread-Regel:
Greife nicht vom Main Thread auf das Model oder die Präsentationskomponenten zu

9.4 Shared data models

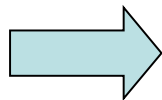
Kompliziertere Programme nutzen auch andere Threads, um Daten zu speichern oder abzufragen, damit die Responsibility nicht beeinträchtigt wird

9.4.1 Thread-safe data models

- Event thread und background threads können ohne Probleme auf dasselbe Model zugreifen, wenn dieses thread-safe ist (z.B. `ConcurrentHashMap`)
- **Alternative:** `CopyOnWriteArrayList` (nur wenn es wesentlich mehr Durchläufe gibt, als Modifikationen)

9.4.2 Split data models

- Table model Klassen, wie `TableModel` und `TreeModel` sind oft selbst Views von anderen Objekten, die die Anwendung verwaltet
- Zwei data models für verschiedene Aufgaben:
 - zur Präsentation
 - für die Anwendung



„split-model design“

9.4.2 Split data models

- Presentation Model ist confined auf den Event Thread
- Shared Model ist thread-safe und kann sowohl vom Event Thread, als auch von anderen Threads aus zugegriffen werden
- Presentation Model registriert listener beim Shared Model, damit ein Update möglich ist (mithilfe eines „Schnappschusses“)

9.4.2 Split data models

Split-model design sollte man in Betracht ziehen, wenn:

- ein Data Model mit mehr als einem Thread geteilt werden muss und
- ein Data Model, das thread-safe ist, nicht zu empfehlen wäre

9.5 Andere Arten von single-threaded subsystems

- Auch andere Systeme als GUIs benutzen thread confinement, z.B. einige native Bibliotheken
- Prinzipiell das gleiche Vorgehen wie bei GUI Frameworks
(`SingleThreadExecutor`, `Future...`)

Zusammenfassung

- GUI Frameworks sind single-threaded
- Präsentationsbezogener Code läuft als Task im Event Thread ab
- Langandauernde Tasks beeinträchtigen die Responsibility und sollten daher in einem Background Thread ausgeführt werden
- SwingWorker o.ä. bietet Support für Cancellation, Progress Indication und Completion Indication