# Nebenläufige und verteilte Programme CS2301


Th. Letschert

TH Mittelhessen Gießen

*University of Applied Sciences*


## Threads und reaktive GUI-Anwendungen

- Reaktiv mit Threads
- Worker-Threads vs UI-Thread
- Ereignis-getriebene Programmierung
- JavaFX Concurrent Tasks

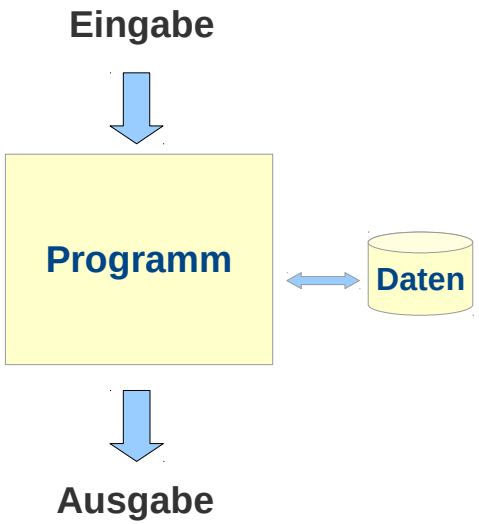
# Reaktive Systeme

**Reaktive Systeme:**

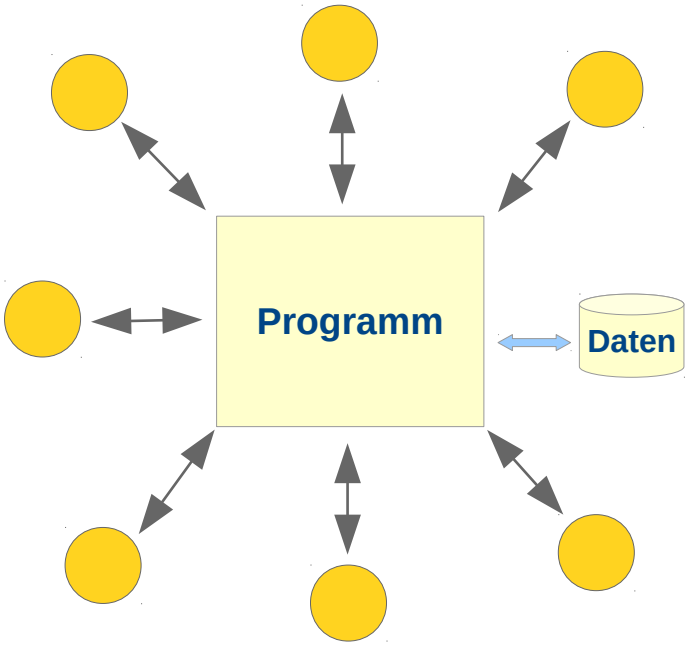
**Systeme die**

- **auf Eingabe von (eventuell vielen) externen Ereignisquellen**
- **schnell (eventuell innerhalb harter Zeitvorgaben)**

**reagieren / antworten müssen.**




*Klassisches System*

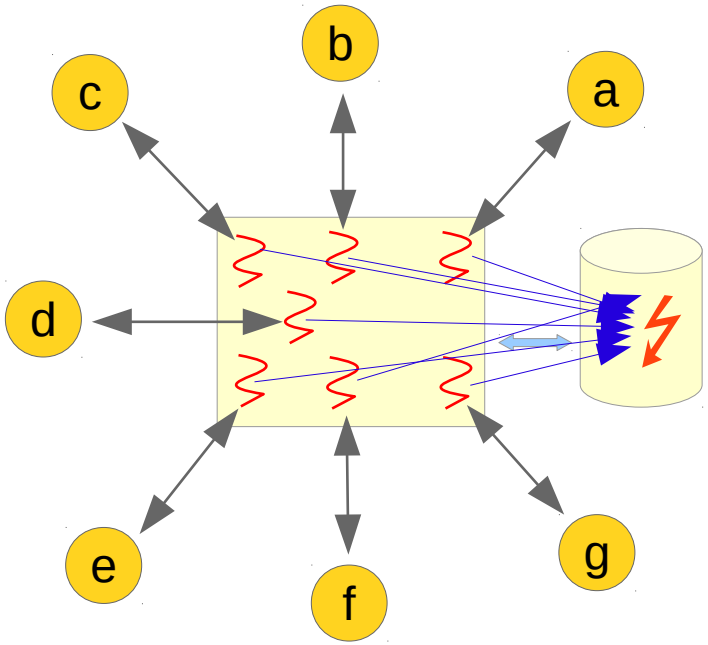



*Reaktives System*

## Reaktive Systeme und Nebenläufigkeit

**Threads**:

- ermöglichen die Reaktivität
- sind Quelle von Interferenzen (Threads kommen sich in die Quere)

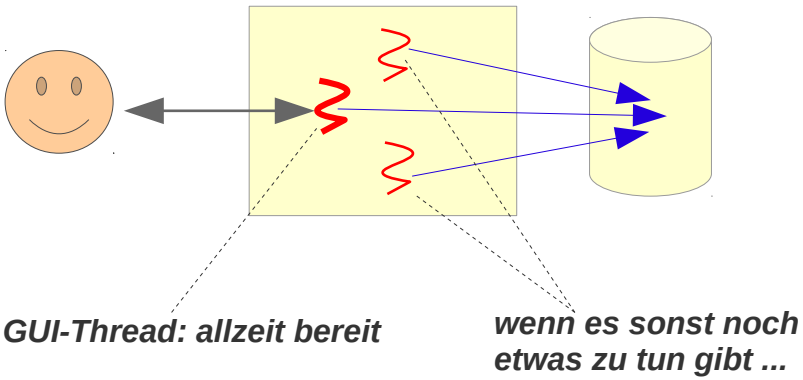


# Anwendungen mit GUI: Reaktive Systeme

## Anwendungen mit Benutzer-Interaktion

- GUI: *Graphical User Interface*
- GUI ~> Reaktives System

  Eine Anwendung mit GUI muss reaktiv sein: sie muss jederzeit auf Eingaben des Benutzers reagieren können



# MVC: Model – View – Control

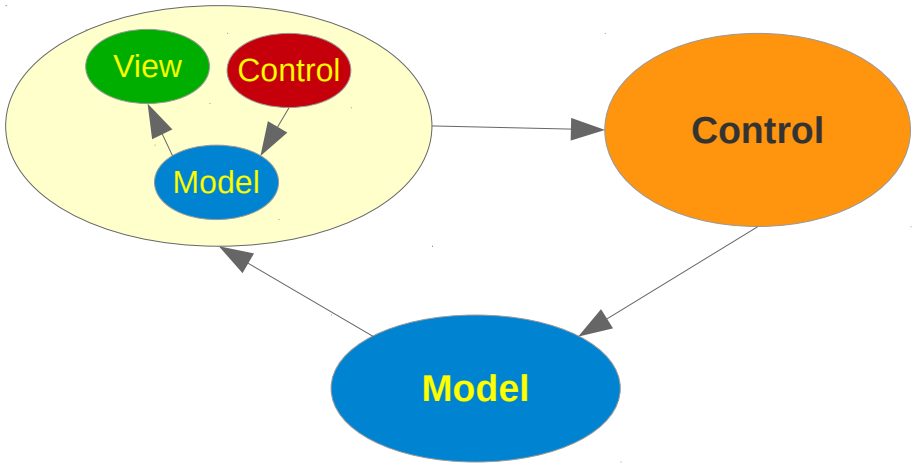
**MVC**: Muster zur Gestaltung von graphisch-interaktiven Anwendungen*

- **Model** Die Daten
- **View** Die Darstellung der Daten
- **Control** Die Verarbeitung der Benutzereingaben/ Modifikation der Daten

## Hierarchische MVC-Struktur

Die Strukturierung in MVC-Komponenten ist in der Regel hierarchisch:

Widgets sind oft View-Komponenten die wieder aus MVC-Komponenten bestehen

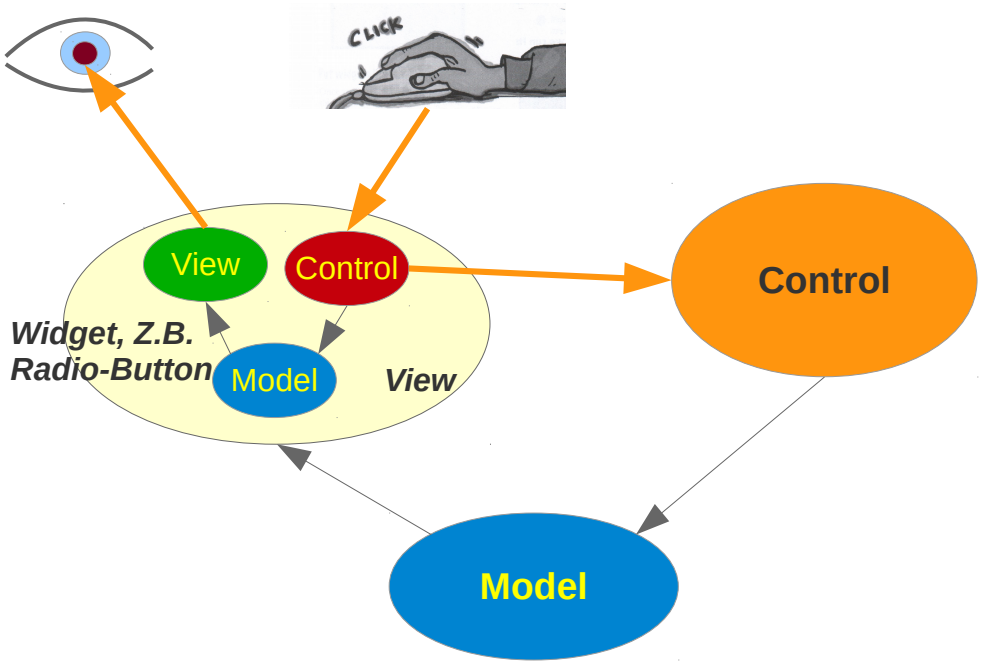



*siehe https://de.wikipedia.org/wiki/Model_View_Controller

# MVC und Reaktiv

## View-Komponenten sollten reaktiv sein

**Eine Aktion des Nutzers sollte unmittelbar Wirkung zeigen**



# Ereignisgetriebene Programmierung

**Reaktiv: Eigenschaft eines Systems (nicht unbedingt nur SW)**

**Das Verhalten eines Systems in Umständen reaktiv**

**Ereignisgetriebene Programmierung: Eine Art reaktives Verhalten realisieren**

**Art der Gestaltung der Software (Paradigma) um reaktives Verhalten zu erreichen**

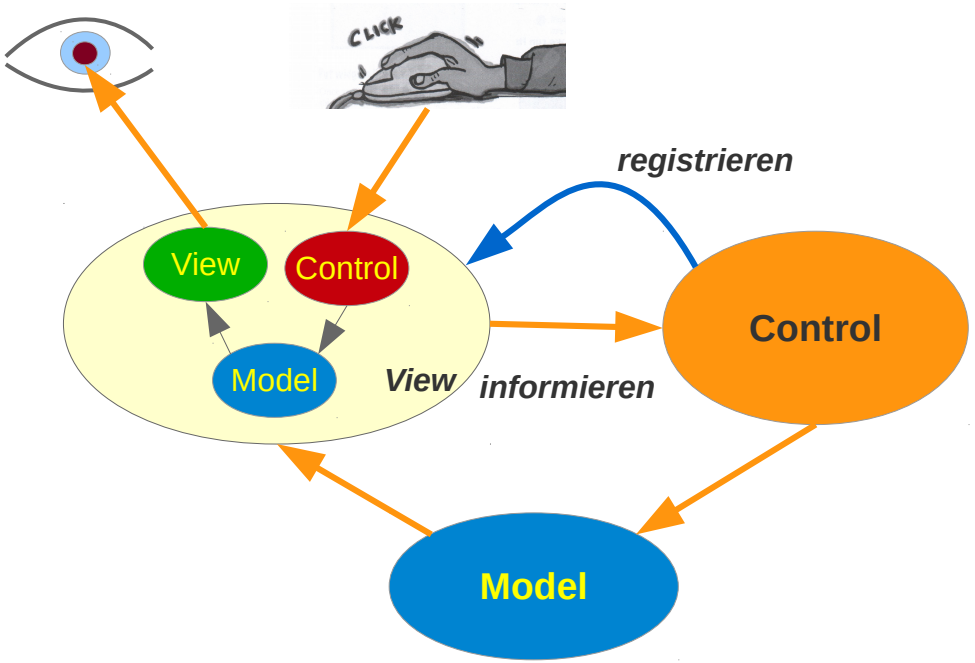
Event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads. Event-driven programming is the dominant paradigm used in graphical user interfaces and other applications (e.g. JavaScript web applications) that are centered on performing certain actions in response to user input.

**Wikipedia: https://en.wikipedia.org/wiki/Event-driven_programming**

# MVC und Ereignisgetrieben

## View-Komponenten sollten reaktiv sein

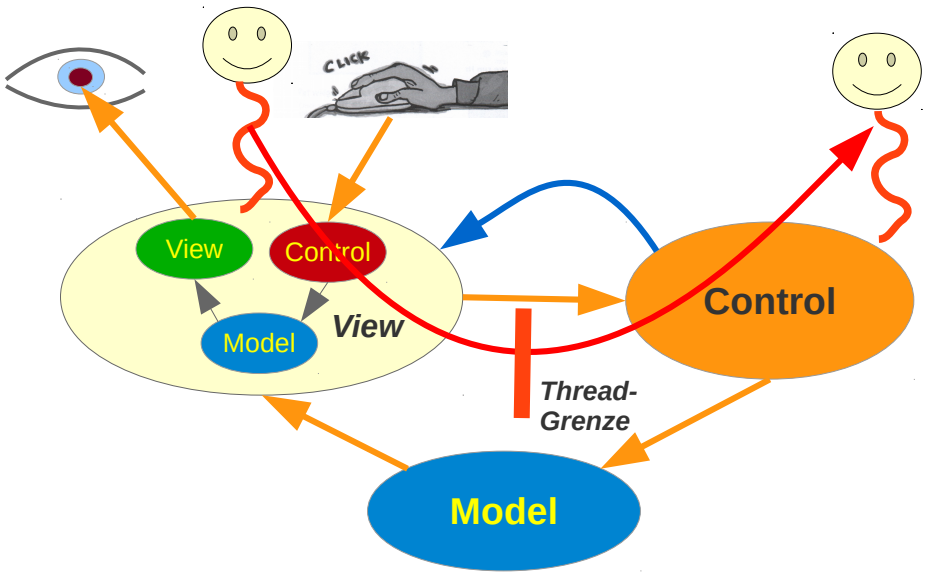
**Die Komponenten leiten Ereignisse nach dem Beobachter-Muster weiter:**
**Bei der Ereignisquelle werden Ereignis-Handler registiert,**
**die das Ereignisziel über ein Ereignis informieren**



# MVC und Ereignisgetrieben

## Ereignis-getrieben: Observer-Muster über Thread-Grenzen hinweg

Event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads. Event-driven programming is the dominant paradigm used in graphical user interfaces and other applications (e.g. JavaScript web applications) that are centered on performing certain actions in response to user input.




***Benutzer-Aktionen werden von einem speziellen UI-Thread bemerkt.***
***Die weitere Ereignis-Behandlung sollte in einem weiteren Thread erfolgen.***

# Kommunikation Thread ~> Thread

## Varianten der Kommunikation über Thread-Grenzen hinweg

- **Callbacks**

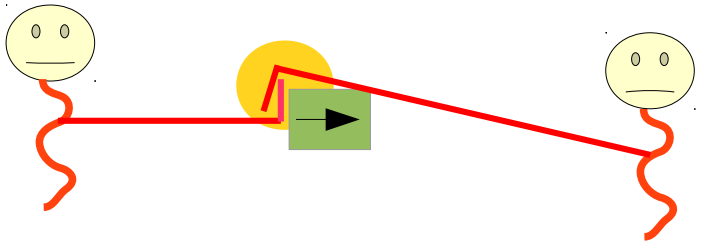
  In machen Sprachen (JavaScript) werden manche Bibliotheksfunktionen asynchron (d.h. in einem eigenen Thread) ausgeführt. Mit einem Callback können Informationen zu dem Thread transportiert werden, der die Funktion aufruft.

  Der Callback wird vom „anderen“ Thread ausgeführt, kann aber auf Ressourcen in „diesem“ Thread zugreifen.

- **Warteschlangen, Futures, ...,**

  In Sprachen mit „richtiger“ Unterstützung der Nebenläufigkeit gibt es weitere Methoden die Thread-Grenze zu Überwinden.

  Letztlich laufen diese alle auf synchronisierte Puffer hinaus.



*Problematik der Kommunikation zwischen Threads: da verhaken sich gerade zwei*

# Beispiel JavaFX

**Wie alle anderen Java-Klassen können JavaFX-Klassen in Scala direkt benutzt werden. Beispiel:**

```
import javafx.application.Application
import javafx.scene.Scene
import javafx.stage.Stage
import javafx.scene.control.{Label, TextField, Button}
import javafx.scene.layout.GridPane

class FXTest extends Application {
  val textfieldInput = new TextField("?")
  val textfieldOutput = new TextField("")
  val buttonCompute = new Button("compute")
  val buttonCancel = new Button("cancel")

  override def start(primaryStage: Stage): Unit = {
    primaryStage.setTitle("Test");

    val view = new GridPane()
    view.add(textfieldInput, 0, 0)
    view.add(buttonCompute, 1, 0)
    view.add(buttonCancel, 2, 0)
    view.add(textfieldOutput, 0, 1)
    view.add(new Label("Result"), 1, 1)
    val scene = new Scene(view);
    primaryStage.setScene(scene);
    primaryStage.show();
  }

}

object FxInScala extends App {
  Application.launch(classOf[FXTest])
}
```

***Definition der statischen Methode launch in Application:***

```
launch(Class<? extends Application> appClass, String... args)
Launch a standalone application.
```

***oder so***

```
object FxInScala {
  def main(args: Array[String]): Unit = {
    Application.launch(classOf[FXTest], args: _*)
  }
}
```

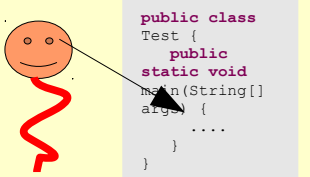
## UI-Thread / Event-Dispatcher-Thread

**Die Zahl der Threads erhöht sich:**

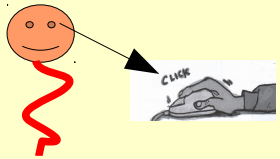
**Der Main-Thread führt den Code aus, der mit dem Aufruf der main-Methode beginnt**

**In interaktiven Anwendungen nach den Start-Up meist ohne Bedeutung**

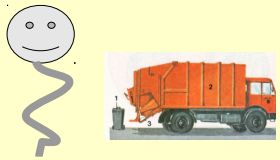
**Der UI-Thread (Event-Dispatcher) führt den Code aus, der mit einer Benutzer-Eingabe an der GUI beginnt.**




*main-Thread: bearbeitet main und alle von dort aufgerufenen Methoden*




*Event-Dispatcher / UI-Thread: bearbeitet interaktive Eingaben, ...*



*Garbage-Collector: sammelt Speicher-Müll*

***Mindestens ein Thread mehr bei interaktiven Anwendungen***

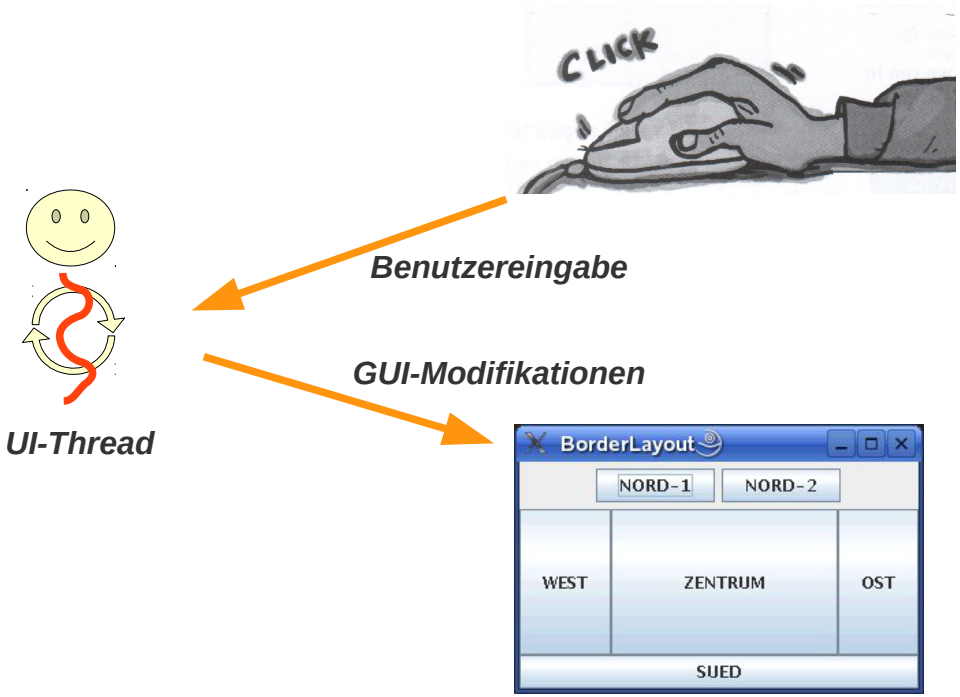
# Single-threaded Event-Dispatching

## *Single-threaded Event Dispatching* Prinzip

**Alle GUIs arbeiten nach diesem Prinzip:**

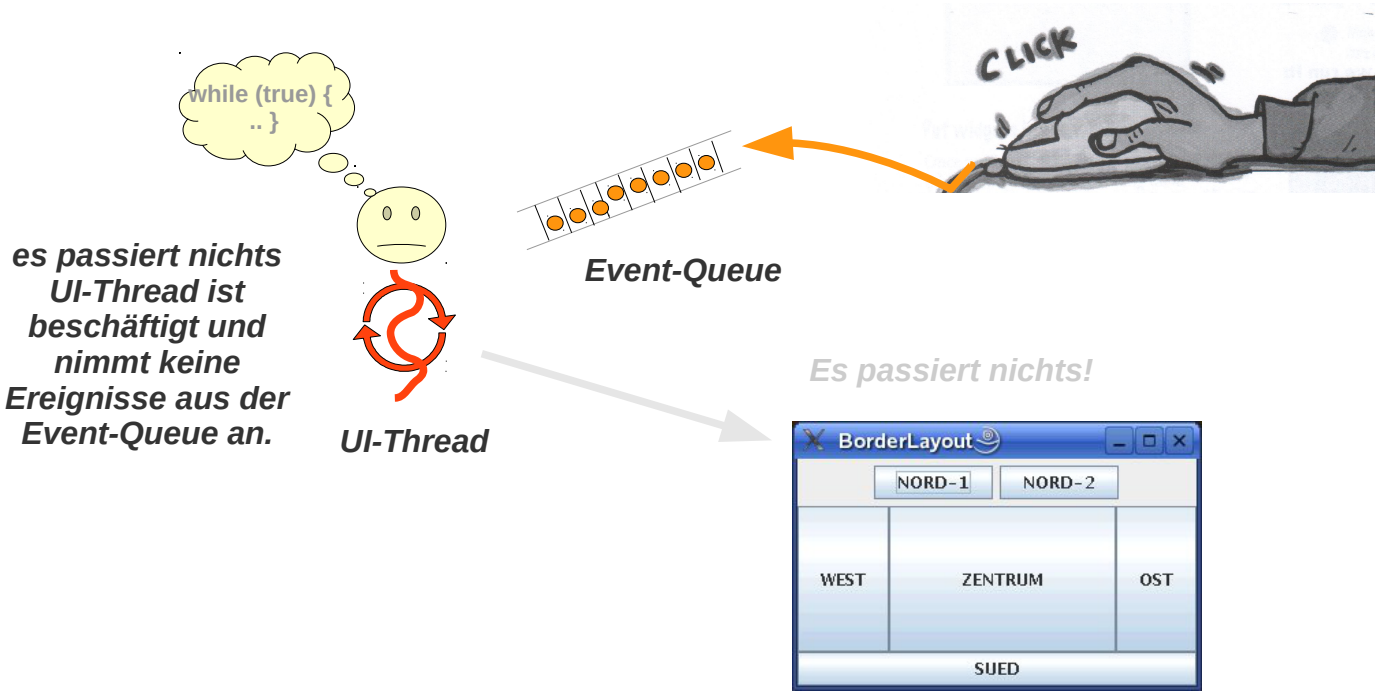
**Ein einziger Thread – der UI-Thread / Event-Dispatcher**

- **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 (Controls) vor**



## Reaktivität garantieren: Hängendes User-Interface vermeiden

**Hängendes UI: Ist der UI-Thread mit langwierigen Operationen beschäftigt, dann hängt die Anwendung**

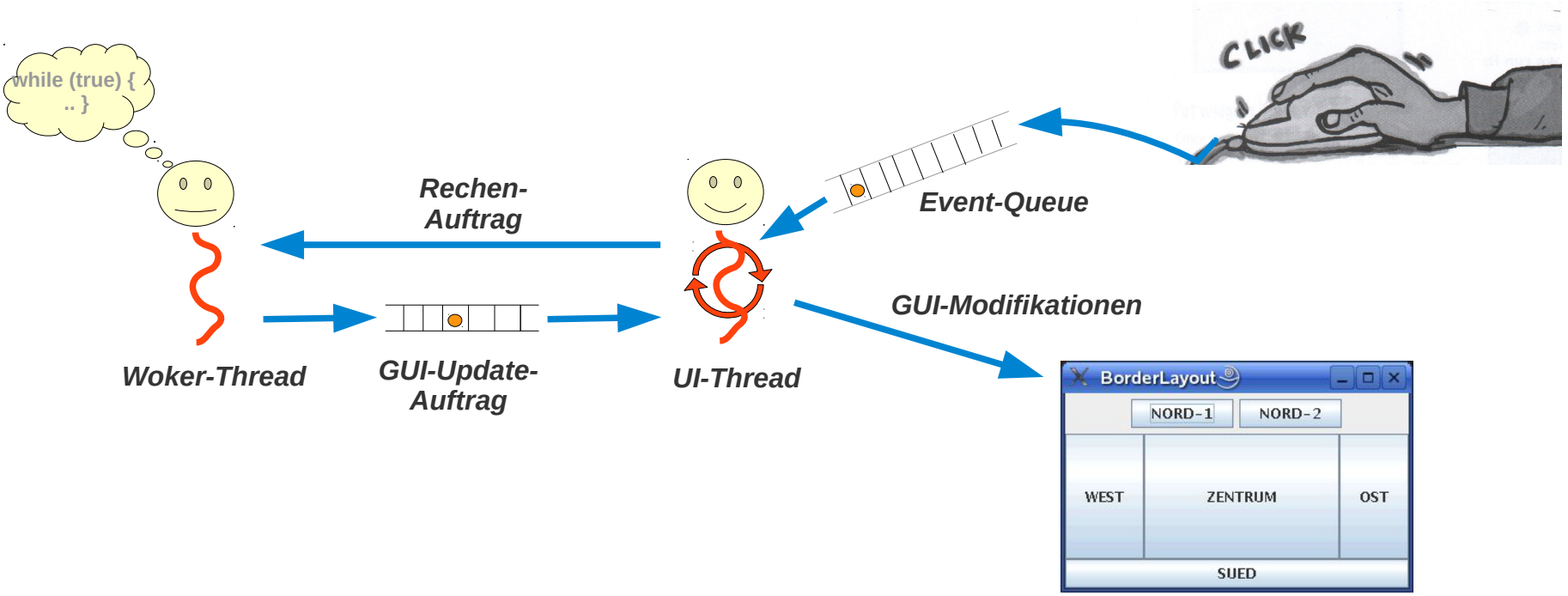


## Reaktivität garantieren: Hängendes UI vermeiden

**Lösung: Langwierige Aktionen in eigenen Worker-Thread auslagern**

**UI-Thread kann weiter Benutzereingaben annehmen**

**Worker-Thread sollte niemals direkt in die GUI-eingreifen (Konflikte!) sondern den UI-Thread mit der GUI-Elemente Änderungen beauftragen**

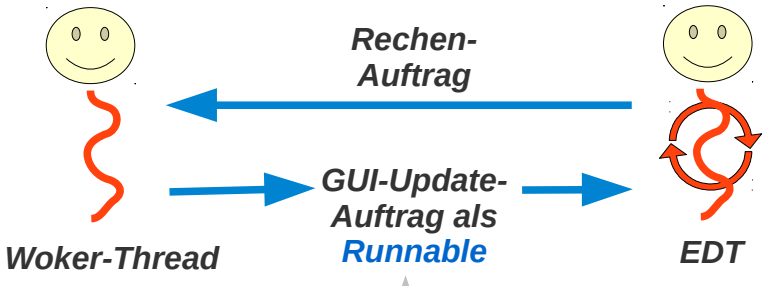


## Interaktion UI-Thread – Worker-Thread

**Swing : UI-Thread = *Event-Dispatcher-Thread EDT***

**Interaktion / Informationsaustausch:**

- **EDT ~> Worker-Thread: Thread im geeigneten Kontext (der die Parameter liefert) starten**
- **Worker-Thread ~> EDT: *invokeLater***




*Der EDT führt das Runnable demnächst bei passender Gelegenheit aus.*

```
SwingUtilities.invokeLater(new Runnable() {
   @Override
   public void run() {
       swingLabel.setText("Work is done");
   }
});
```

*Ausführlich (Java)*

```
SwingUtilities.invokeLater(
      () -> swingLabel.setText("Work is done");
);
```

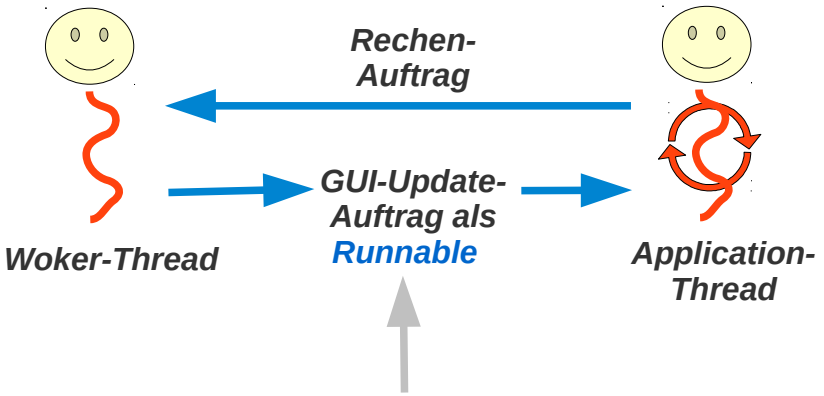
*.. oder in Java-8 Kurznotation (Runnable ist ein Functional-Interface)*

## Interaktion UI-Thread – Worker-Thread

**FX :** UI-Thread = *Application-Thread*

**Interaktion / Informationsaustausch:**

- UI-Thread ~> Worker-Thread: Thread im geeigneten Kontext (der die Parameter liefert) starten
- Worker-Thread ~> UI-Thread:




*… also praktisch genau wie in Swing: Übergabe eines Runnables, das die Aktualisierung der GUI-Komponente vornimmt.*

```
Platform.runLater(
      new Runnable {
            override def run() {
              fxTextfield.setText("Work is done!");
            }
});
```

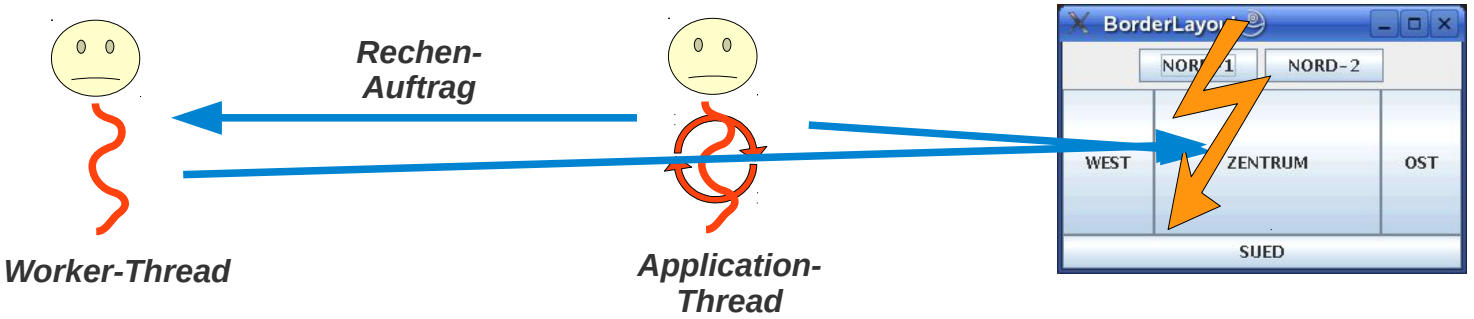
*Ausführlich (in Scala)*

```
Platform.runLater(
  () => fxTextfield.setText("Work is done!");
);
```

*Oder in Scala ab 2.12 mit Funktionen an Stelle einer Instanz eines SAM-Interfaces.*

## Interaktion UI-Thread – Worker-Thread

**FX :** UI-Thread = *Application-Thread*




*Der Worker-Thread kann auch direkt auf GUI-Elemente zugreifen.*
*Es sollte es aber nicht tun! – Zwei Threads kommen sich ansonsten unsynchronisiert in die Quere.*

# Interaktion UI-Thread – Worker-Thread

## Beispiel: mit explizit verwaltetem Worker- (Faktorisierer-) Thread

```
var factorizingThread: Option[Thread] = None

buttonCompute.setOnAction(
  (ae: ActionEvent) => {
    if (factorizingThread.isDefined) {
      tfOutput.setText("I'm busy!")
    } else {
      tfOutput.setText("")
      factorizingThread = Some(
        new Thread(() => {
          try {
            val input: Long = Integer.parseInt(tfInput.getText())
            val res = factors(input)
            res match {
              case Success(l)                 => Platform.runLater( () => { tfOutput.setText(l.toString) })
              case Failure(CanceledException) => Platform.runLater( () => { tfOutput.setText("Canceled") })
              case Failure(t)                 => Platform.runLater( () => { tfOutput.setText("Error " + t) })
            }
          } catch {
            case e: Exception => Platform.runLater( () => { tfOutput.setText("Error " + e) })
          }
          factorizingThread = None
        })
      )
      factorizingThread.get.start()
    }
  })
```

*Test ob bereits eine Faktorisierung läuft*

# Package javafx.concurrent

## Nebenläufigkeit etwas moderner

**Eine direkte Verwendung von Threads**

- **durch Anwendungsprogramme (und ihre Entwickler)**
- **wird heute nicht mehr empfohlen**

**Das Package *javafx.concurrent* liefert Komponenten**

- **für die Realisation von Nebenläufigkeit in interaktive Anwendungen**
- **auf etwas höherem Abstraktionsniveau**

**Das Package *java.util.concurrent***

- **liefert die Basis von *javafx.concurrent***
- ***javafx.concurrent* betrachten wir darum erst später etwas genauer nach der Beschäftigung mit *java.util.concurrent***

**Package javafx.concurrent Description**

Provides the set of classes for javafx.task.

This package provides the ability to run application code on threads other than the JavaFX event dispatch thread. The ability to control the execution and track the progress of the application code is also provided.

# Worker

## Interface *Worker*

**Ein Worker ist ein Objekt, das eine Arbeit im Hintergrund ausführt.**

**Achtung: Ein *Worker* ist kein Thread!**

- **Ein *Worker* wird durch einen oder mehrere Threads ausgeführt**
- **Mit den Threads, die dem *Worker* „Leben einhauchen“, hat die Anwendung (und ihre Entwickler) i.A. und (wenn sie will) nichts zu tun**

  **Moderne Anwendungen vermeiden möglichst jeden direkten Kontakt mit Threads**

**Worker gibt es in drei Varianten (Klassen die das Worker-Interface implementieren):**

- **Task:**

  **Eine asynchrone* Berechnung die einmal ausgeführt wird**
- **Service:**

  **Eine wiederverwendbare asynchrone* Berechnung**

  **Realisiert als Task mit einer „Verwaltungshülle“**
- **ScheduledService:**

  **Eine regelmäßig ausgeführte asynchrone* Berechnung**

  **Realisiert als Task der regelmäßig ausgeführt wird.**

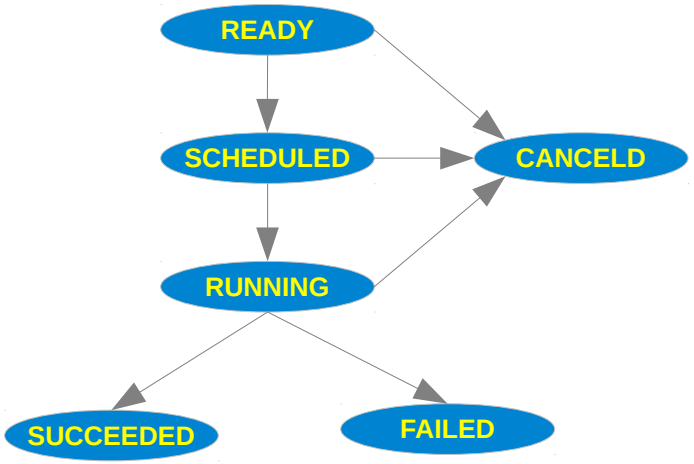
***asynchron: entkoppelt in einem anderen Thread**

# Worker

## Eigenschaften als *Properties*

**Ein Worker ist in einem Zustand:**

- `Worker.State.READY`
- `Worker.State.SCHEDULED`
- `Worker.State.RUNNING`
- `Worker.State.CANCELED`
- `Worker.State.SUCCEEDED`
- `Worker.State.FAILED`



# Beispiel Task

**Einfaches Beispiel: Worker = javafx.concurrent.Task**

**ohne Ergebnis,**

**der direkt auf GUI-Komponenten zugreift.**

**Ausführung mit selbst erzeugtem Thread**

```
val tfOutput = new TextField("")
val buttonCompute = new Button("compute")
```

*Ein Task,*
*.. kein großer Unterschied zu oben*

```
val task = new Task[Void] {
  override def call(): Void = {
    Thread.sleep(5000) // do some hard work
    Platform.runLater(
       new Runnable {
         override def run() {
           tfOutput.setText("Mission completed!");
         }
       });
    null
  }
}
```

*Hier wird ein Thread gestartet, um den Task auszuführen. Das ist möglich aber nicht die bevorzugte Methode.*
*Besser nutze einen Threadpool.*
*(Dazu später mehr)*

```
buttonCompute.setOnAction(
    new EventHandler[ActionEvent] {
     def handle (ae: ActionEvent): Unit = {
       new Thread(
           task
       ).start()
     }
    }
  )
```

# Beispiel Task

**Einfaches Beispiel: Worker = javafx.concurrent.Task**

**mit Ergebnis,**

**der nicht direkt auf GUI-Komponenten zugreift**

**Ausführung mit selbst erzeugtem Thread**

```
val tfOutput = new TextField("")
val buttonCompute = new Button("compute")

val task = new Task[Int] {
  override def call(): Int = {
    Thread.sleep(5000) // do some hard work
    42 // finally find the answer
  }
}
```

*Der Task bekommt einen OnSucceeded-Handler, der das Ergebnis im UI setzt.*

*Platform.runLater ist nicht mehr nötig. Der Handler wird vom Application-Thread ausgeführt.*

```
task.setOnSucceeded(new EventHandler[WorkerStateEvent]{
  def handle(event: WorkerStateEvent): Unit = {
    val result = event.getSource().getValue
    tfOutput.setText(""+result)
  }
})
```

```
buttonCompute.setOnAction(
    new EventHandler[ActionEvent] {
     def handle (ae: ActionEvent): Unit = {
       new Thread(
           task
         ).start()
     }
    }
  )
```

# Beispiel Task

**Einfaches Beispiel: Task**

- mit Ergebnis
- der nicht direkt auf GUI-Komponenten zugreift
- Ausführung in Thread-Pool (Executor)

```
val tfOutput = new TextField("")
val buttonCompute = new Button("compute")

val task = new Task[Int] {
  override def call(): Int = {
    Thread.sleep(5000) // do some hard work
    42
  }
}

task.setOnSucceeded(new EventHandler[WorkerStateEvent]{
  def handle(event: WorkerStateEvent): Unit = {
    val result = event.getSource().getValue
    tfOutput.setText(""+result)
  }
})
```

```
val NumberOfcores = Runtime.getRuntime().availableProcessors()
val executor: ExecutorService = Executors.newFixedThreadPool(NumberOfcores)

buttonCompute.setOnAction(
    new EventHandler[ActionEvent] {
     def handle (ae: ActionEvent): Unit = {
       executor.submit(task)
     }
    }
  )
```

```
override
def start(primaryStage: Stage): Unit = {
  ...

  primaryStage.setOnCloseRequest(new EventHandler[WindowEvent] {
        def handle(we: WindowEvent): Unit = {
            executor.shutdown()
        }
    });
  ...
}
```

*Executor (Thread-Pool) starten*

*... und Runterfahren nicht vergessen*

## Java-Util-Concurrent-Executor

**Berechnung in einem Thread-Pool (Executor) starten**

**Details später.**

```
import java.util.concurrent.{Executors, ExecutorService}

...

val NumberOfcores = Runtime.getRuntime().availableProcessors()
val executor: ExecutorService = Executors.newFixedThreadPool(NumberOfcores)
```

*Executor (Thread-Pool) erzeugen und starten*

```
...

executor.submit(task)
```

*Task im Executor ausführen*

```
...

primaryStage.setOnCloseRequest(new EventHandler[WindowEvent] {
     def handle(we: WindowEvent): Unit = {
               executor.shutdown()
     }
})
```

*Executor stoppen*

# Task: Ergebnis verwenden

## javafx.concurrent.Task Ergebnis verwenden

```
class FactorizingTask(nS: String) extends Task[List[Long]] {
  override def call(): List[Long] = {
    ...
  }
}
```

*Faktorisierer-Task*

```
task.setOnSucceeded( (event: WorkerStateEvent) => {
   val result = event.getSource().getValue
   Platform.runLater( tfOutput.setText(""+result) )
})

task.setOnCancelled( (event: WorkerStateEvent) => {
    Platform.runLater( tfOutput.setText("canceled") )
})

task.setOnFailed( (event: WorkerStateEvent) => {
    Platform.runLater( tfOutput.setText("Failed: " +
                          event.getSource().getException) )
})

executor.submit(task)
```

*Faktorisierer erzeugen*

*„Callbacks“ anhängen für*

- *Erfolg*
- *Abbruch*
- *Fehler*

*Faktorisierer starten*

*Scala < 2.12:*
*Zur Vereinfachung kommen implizite „SAM-Konversionen“ zum Einsatz*

```
implicit def eventHandler[E <: Event](handler: E => Unit): EventHandler[E] =
  new EventHandler[E]{
    override def handle (ae: E): Unit = {
      handler(ae)
    }
}
```

## Task Abbruch

**Ein Task sollte mit isCanceled regelmäßig prüfen, ob er gecancelt wurde.**

**Alternativ kann auch das Interrupted-Flag geprüft werden**

**Empfehlenswert in Code von dem nicht klar ist, ob er in einem FX-Task ausgeführt wird**

```
Task<Integer> task = new Task<Integer>() {
    @Override protected Integer call() throws Exception {
        int iterations;
        for (iterations = 0; iterations < 100000; iterations++) {
            if (isCancelled()) {
                break;
            }
            System.out.println("Iteration " + iterations);
        }
        return iterations;
    }
};
```

***Beispiel aus der javaFx API***

# Task Abbruch, Beispiel

## Abbrechbarer Task mit Ergebnis

```
class FactorizingTask(nS: String) extends Task[List[Long]] {
  override def call(): List[Long] = {
    val n: Long = Integer.parseInt(nS) // may throw an exception
    if (n < 2) throw new IllegalArgumentException()
    else {
      try {
        Range.Long(2L, n/2+1, 1)
         .filter( (i: Long) => {
            if (isCancelled()) {
              throw CanceledException
            }
            n%i == 0 && isPrime(i);
         })
         .toList
      } catch {
        case CanceledException => List()
      }
    }
  }
}
```

***damit der laufende Thread tatsächlich beendet wird, muss er auf den Interrupt reagieren!***

```
buttonCancel.setOnAction(
    (ae: ActionEvent) => {
      if (!factorizingTask.isDefined) {
        tfOutput.setText("Nothing to be canceled")
      } else {
        factorizingTask.get.cancel()
      }
    }
  )
```

# Task Abbruch, Beispiel

## Abbrechbaren Task managen

```
var factorizingTask: Option[FactorizingTask] = None

buttonCompute.setOnAction(
    (ae: ActionEvent) => {
      if (factorizingTask.isDefined) {
        tfOutput.setText("I'm busy!")
      } else {
        val task = new FactorizingTask(tfInput.getText())
        task.setOnSucceeded( ... )
        task.setOnCancelled( ... )
        task.setOnFailed( ... )
        executor.submit(task)
        factorizingTask = Some(task)
      }
    }
 )

buttonCancel.setOnAction(
    (ae: ActionEvent) => {
      if (!factorizingTask.isDefined) {
        tfOutput.setText("Nothing to be canceled")
      } else {
        factorizingTask.get.cancel(true)
        factorizingTask = None
      }
    }
  )
```

# Service

**Die Klasse *javafx.concurrent.Service* vereinfacht den Umgang mit Threads**

**Beispiel (1):**

```
class FactorizingService extends Service[List[Long]] {
  val inputProp : StringProperty = new SimpleStringProperty(this, "")

  def setInput(input: String): Unit = { inputProp.set(input) }
  def getInput(): String = inputProp.get()

  setExecutor(executor)

  override def createTask() : Task[List[Long]] = {
    val nS = getInput()
    new Task[List[Long]] {
      updateProgress(0, 1)
      updateMessage("")
      override def call(): List[Long] = {
        val n: Long = Integer.parseInt(nS)
        if (n < 2) throw new IllegalArgumentException()
        else {
           ...
        }
      }
    }
  }

}
```

*Die Eingabe wird zur Property, die von aussen gesetzt wird.*

*Der* `executor` *ist der Threadpool auf dem der Task ausgeführt werden soll.*

*Der Task wird wie wie oben erzeugt.*

# Service

**Die Klasse *javafx.concurrent.Service* vereinfacht den Umgang mit Threads**

**Beispiel (2):**

```
val factorizingService = new FactorizingService

buttonCompute.setOnAction(
    (ae: ActionEvent) => {
      if (factorizingService.isRunning()) {
        tfOutput.setText("I'm busy!")
      } else {
        tfOutput.setText("")
        factorizingService.setInput(tfInput.getText)
        factorizingService.restart()
        progressbar.progressProperty().bind(factorizingService.progressProperty())
        tfMsg.textProperty().bind(factorizingService.messageProperty())

        factorizingService.setOnSucceeded( ... )
        factorizingService.setOnCancelled( ... )
        factorizingService.setOnFailed( ... )
      }
    }
 )
```

*Das Thread-Management wird von der Service-Klasse übernommen.*

*`restart` statt eigener Thread-Erzeugung*

# Service

**Die Klasse *javafx.concurrent.Service* vereinfacht den Umgang mit Threads**

**Beispiel (3):**

```
buttonCancel.setOnAction(
    (ae: ActionEvent) => {
      if (!factorizingService.isRunning()) {
        tfOutput.setText("Nothing to be canceled")
      } else {
        factorizingService.cancel()
        ()
      }
    }
  )
```

*Das Thread-Management wird von der Service-Klasse übernommen.*

*`cancel` und `isRunning` statt eigener Thread-Überwachung*

# Service

**Die Klasse *javafx.concurrent.Service* vereinfacht den Umgang mit Threads**

**Beispiel (4):**

```
  val NumberOfcores = Runtime.getRuntime().availableProcessors()
  val executor: ExecutorService = Executors.newFixedThreadPool(NumberOfcores)

  def submitToExecutor(runnable: Runnable): Future[_] = executor.submit(runnable)

  def shutdownExecutor : Unit = executor.shutdown
```

```
class Factorizing_Application extends Application {

  override def start(primaryStage: Stage): Unit = {
    val startView = new StartView
    val scene = new Scene(startView)
    primaryStage.setScene(scene);

    primaryStage.setOnCloseRequest(new EventHandler[WindowEvent] {
          def handle(we: WindowEvent): Unit = {
              shutdownExecutor
          }
     })

    primaryStage.show();
  }

}
```

*Der Threadpool wird weiterhin selbst verwaltet.*