ISA
Institut für
SoftwareArchitektur

Nebenläufige und verteilte Programme CS2301

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Reactive Programmierung

- Datenfluss-Paradigma der reaktiven Programmierung
- reaktive Programmierung mit JavaFX

Reaktive Systeme und Reaktive Programmierung

Reaktives System

System das auf die Eingabe von (vielen) externen Ereignisquellen (schnell) reagieren muss

Reaktive Programmierung

Programmierstil / Programmierparadigma bei dem

- **Ereignisse**
- **und deren Verknüpfung / Weiterleitung**

durch die Sprache / ein Framework unterstützt wird.

Reaktive Systeme und reaktive Programmierung

Die reaktive Programmierung liefert (Software-) Abstraktionen die
die Realisation reaktiver Systeme erleichtern

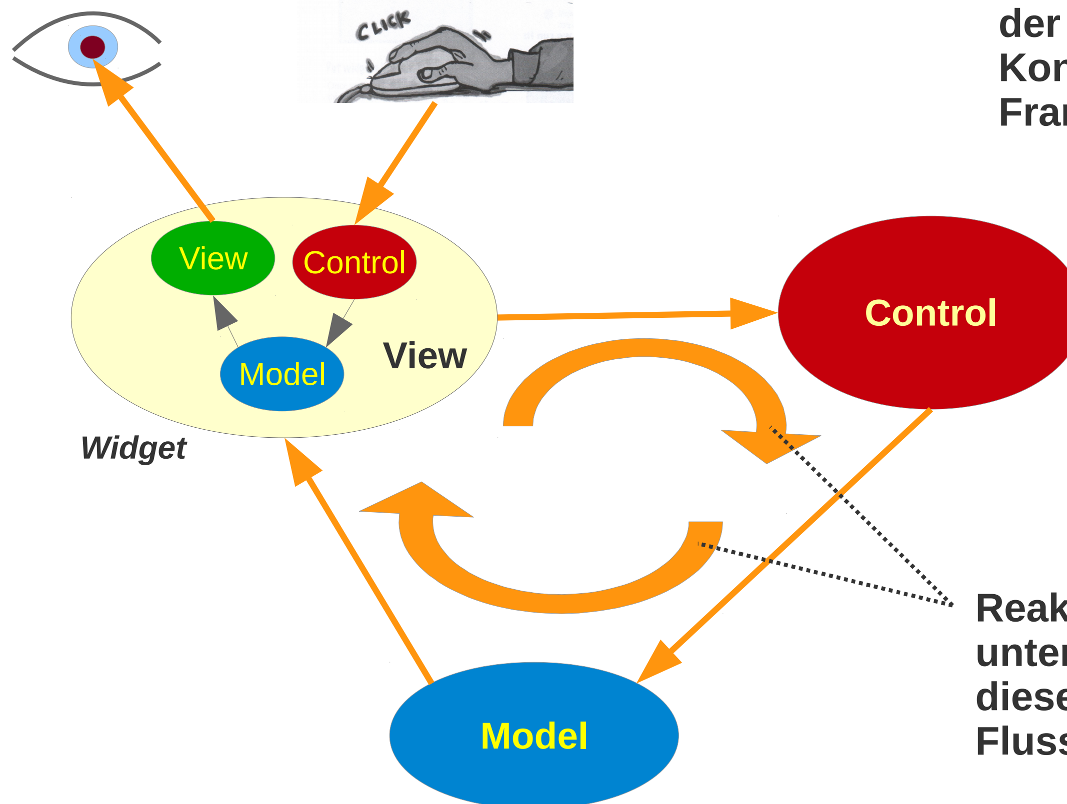
> Reactive programming tackles issues posed by event-driven applications by providing abstractions to express programs as reactions to external events and having the language automatically manage the flow of time (by conceptually supporting simultaneity), and data and computation dependencies.
>
> E. Bainomugisha, A. L. Carreton, T. van Cursem, S. Mostinckk, W. de Meuter:
> *A Survey on Reactive Programming;*
> ACM Computing Surveys, Volume 45 Issue 4, August 2013

Reaktive Programmierung und GUIs

GUIs sind reaktive Anwendungen

Reaktive Programmierung sollte hier eingesetzt werden

MVC + Beobachtermuster
Richtlinie zur Implementierung
der Daten- / Ereignisse in üblichen
Kontrollfluss-orientierten Sprachen /
Frameworks



Reaktive Programmierung
unterstützt die Realisation
dieses Ereignis-/ Daten-
Flusses direkt(er).

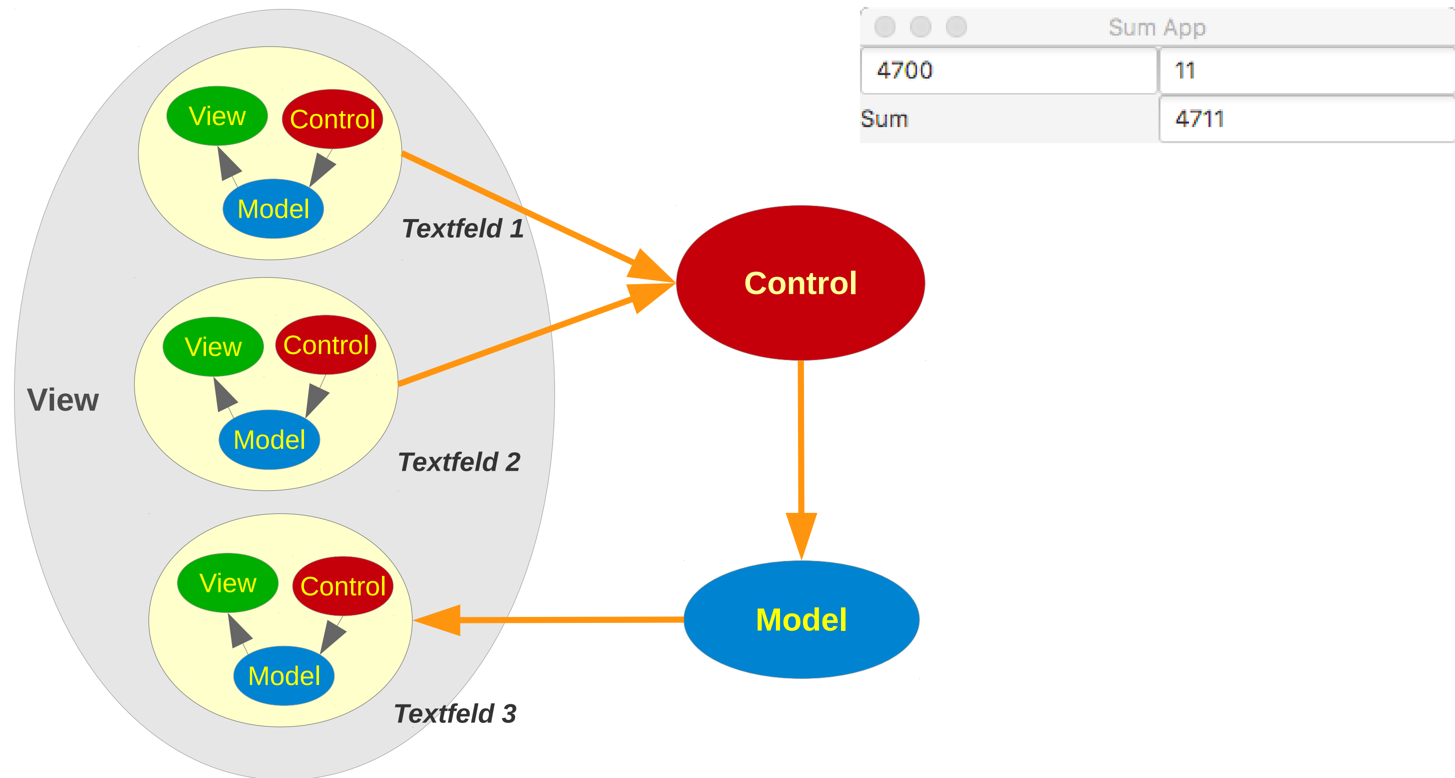
Reaktiv mit „automatisiertem“ Beobachter-Muster

Ereignis-getriebene Programmierung:

SW wird entsprechend dem Beobachtermuster organisiert.

Die Benachrichtigungen erfolgen „irgendwie automatisch“

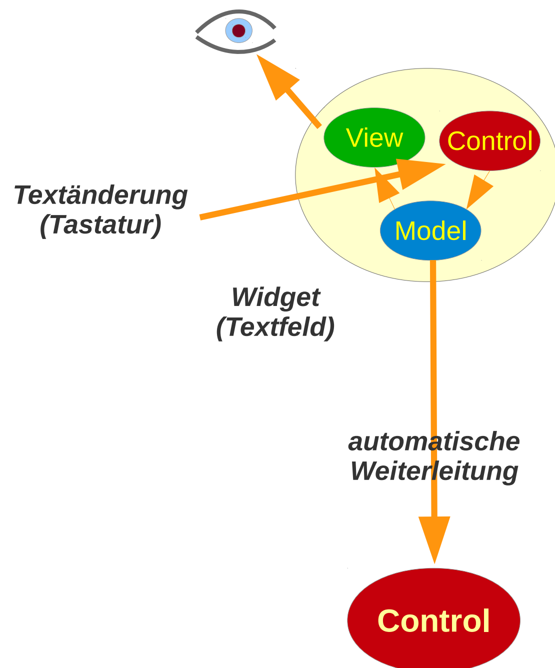
Beispiel: Ein Ausgabe-Textfeld zeigt die Summe des Inhalts von zwei Eingabe-Textfeldern an



Reaktiv mit „automatisiertem“ Beobachter-Muster

Beispiel: Ein Ausgabe-Textfeld zeigt die Summe des Inhalts von zwei Eingabe-Textfeldern an

Reaktive / Beobachtbare Widgets, hier Textfelder




```
val tfInput_1 = new TextField("")

tfInput_1.textProperty().addListener(
  new ChangeListener[String]{
    override def changed( obs: ObservableValue[_ <: String],
                          oldValue: String,
                          newValue: String) = {
      SumControl.set_a(newValue)
    }
  })

// oder kurz:
tfInput_1.textProperty().addListener(
  (obs , oldValue,  newValue) => {
      SumControl.set_a(newValue)
  })
```

**Ein Textfeld hat eine beobachtbare TextProperty.
Beobachter können sich dort registrieren und werden automatisch über Änderungen informiert.**

JavaFx: Observable und ObservableVaue

Interface javafx.beans.Observable

Ein *Observable* enthält Inhalte über deren Veränderung Listener (Beobachter) informiert werden

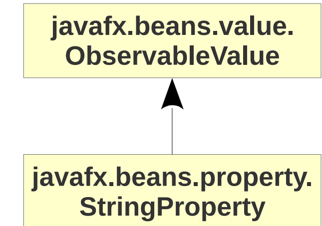
Interface javafx.beans.InvalitationListener

Ein InvalitationListener wird über veränderte Inhalte eines Observable informiert.

JavaFx: Properties

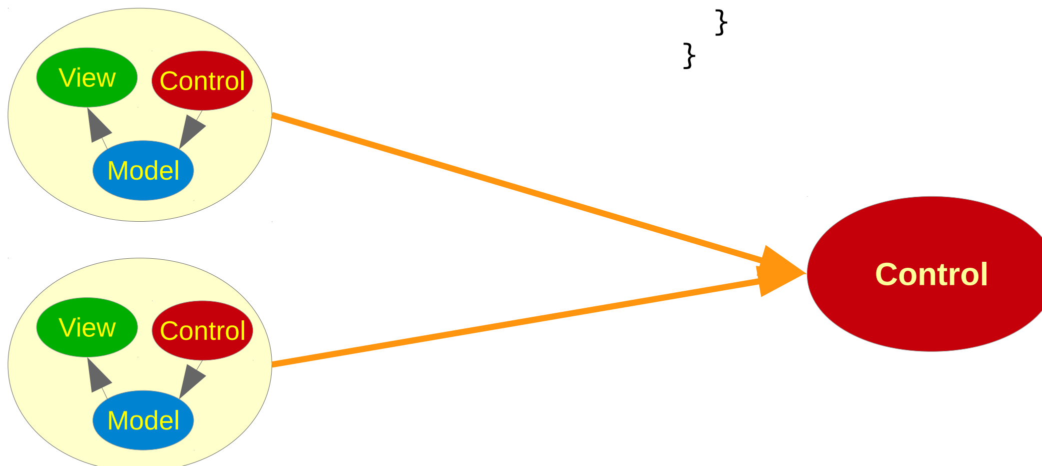
Properties

sind durch das Interface javafx.beans.property.Property<T> beschriebene beobachtbare Komponenten mit ein paar zusätzlichen Eigenschaften.
Ein Textfeld hat eine textProperty vom Typ StringProperty




```
tfInput_1.textProperty().addListener(
  (obs , oldValue,  newValue) => {
      SumControl.set_a(newValue)
  })

tfInput_2.textProperty().addListener(
  (obs , oldValue,  newValue) => {
      SumControl.set_b(newValue)
  })
```

```
object SumControl {
  def set_a(str: String): Unit = try {
    val a = str.toInt
    SumModel.setA(a)
  } catch {
    case _:NumberFormatException => /* ignore */
  }
  def set_b(str: String): Unit = try {
    val b = str.toInt
    SumModel.setB(b)
  } catch {
    case _:NumberFormatException => /* ignore */
  }
}
```

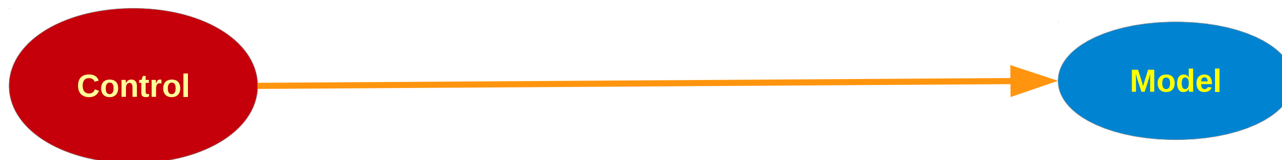


JavaFx: Unterstützung bei der Umsetzung des Beobachtermusters

ObsevableBase : Hilfsklasse zur Definition eines Observables

```
object SumControl {
  def set_a(str: String): Unit = try {
    val a = str.toInt
    SumModel.setA(a)
  } catch {
    case _:NumberFormatException => /* ignore */
  }
  def set_b(str: String): Unit = try {
    val b = str.toInt
    SumModel.setB(b)
  } catch {
    case _:NumberFormatException => /* ignore */
  }
}
```

```
object SumModel extends ObservableValueBase[Int] {
  private var value_a = 0
  private var value_b = 0

  def setA(v: Int) : Unit = {
    value_a = v
    fireValueChangedEvent()
  }
  def setB(v: Int) : Unit = {
    value_b = v
    fireValueChangedEvent()
  }

  def getValue(): Int = value_a + value_b

}
```



Das Model ist ein beobachtbarer Int-Wert

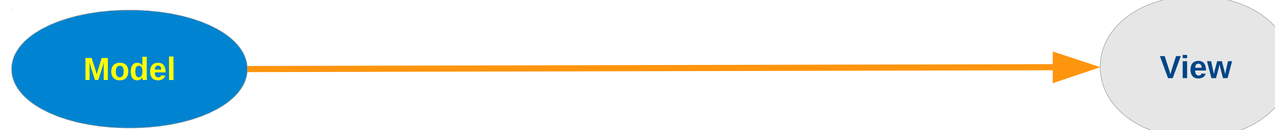
JavaFx: Unterstützung bei der Umsetzung des Beobachtermusters

ChangeListener :

Interface zur Definition eines Beobachters

```
class SumView extends GridPane with ChangeListener[Int] {
  val tfInput_1 = new TextField("")
  val tfInput_2 = new TextField("")
  val tfOutput = new TextField("")

  add(tfInput_1, 0, 0)
  add(tfInput_2, 1, 0)
  add(tfOutput, 1, 1)
  add(new Label("Sum"), 0, 1)

  tfInput_1.textProperty().addListener(
    (obs , oldValue,  newValue) => {
        SumControl.set_a(newValue)
    })

  tfInput_2.textProperty().addListener(
    (obs , oldValue,  newValue) => {
        SumControl.set_b(newValue)
    })

  override def changed( obs: ObservableValue[_ <: Int],
                        oldValue: Int,
                        newValue: Int) = {
    tfOutput.setText(""+newValue)
  }

}
```

```
object SumModel extends ObservableValueBase[Int] {
  private var value_a = 0
  private var value_b = 0

  def setA(v: Int) : Unit = {
    value_a = v
    fireValueChangedEvent()
  }
  def setB(v: Int) : Unit = {
    value_b = v
    fireValueChangedEvent()
  }

  def getValue(): Int = value_a + value_b

}
```



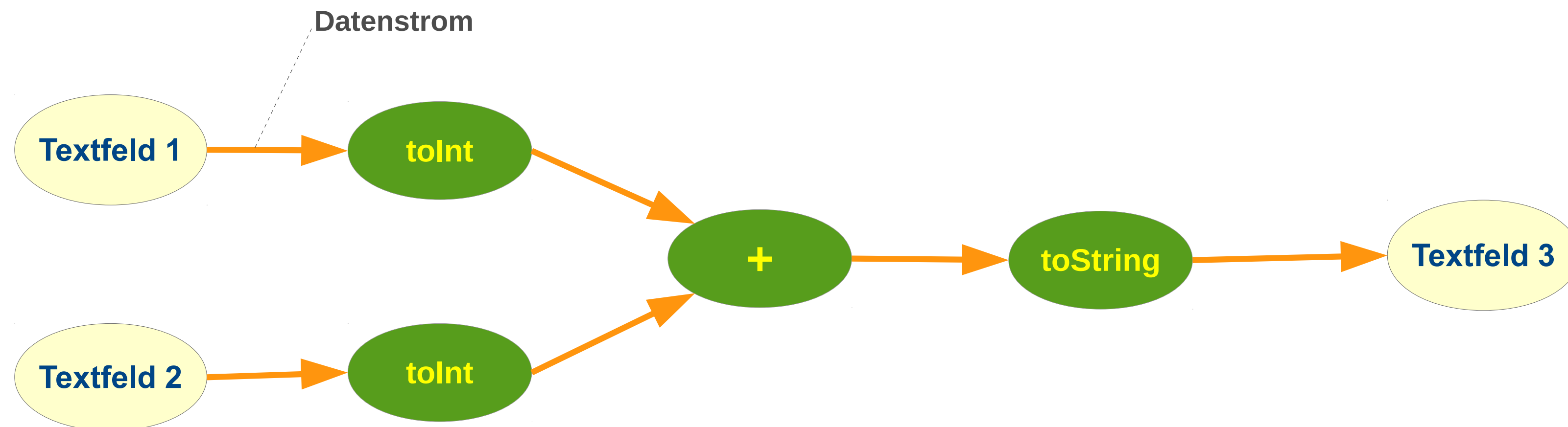
Reaktive Programmierung ~ Datenfluss

Kontrollfluss

Der Programmcode spezifiziert den Kontrollfluss, der Datenfluss ergibt sich implizit
Übliche Art der Programmierung mit Anweisungen, Prozeduren, …

Datenfluss

Der Programmcode spezifiziert den Datenfluss, der Kontrollfluss ergibt sich implizit
Beispiel: Datenfluss der Summations-Anwendung:



Reaktive Programmierung muss (!) das Datenfluss-Paradigma unterstützen

JavaFX und das Datenfluss-Paradigma

Properties

sind durch das Interface javafx.beans.property.Property<T> beschriebene beobachtbare Komponenten mit ein paar Eigenschaften, die über die von *ObservableValue* hinausgehen:

- ***Properties* können gebunden werden**
  Das Binden ist eine „Automatisierung“ des Beobachtens.
- ***Properties* können**
  Auskunft über ihren Namen und die umfassende Komponente geben

Bindings

Ein *Binding* ist eine beobachtbare Komponenten deren Wert / Zustand aus dem anderer Komponenten automatisch berechnet wird.

Ein Binding wird aus Observables konstruiert und ist selbst beobachtbar.

Ein Binding von unterschiedlichen Werten vom Typ T ergibt ein Observable<T>

Beispiel:

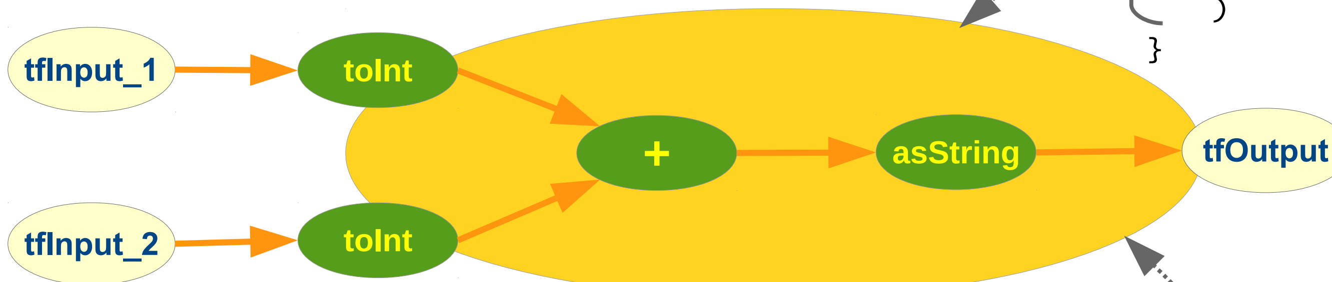
- **a, b: Integer-Properties (also ObservableValues)**
- **a und b werden mit add gebunden**
- **ergeben einen beobachtbaren Wert: ihre Summe**

JavaFX und das Datenfluss-Paradigma

JavaFX erlaubt die Definition von Datenfluss-Graphen

```
class StringToIntBinding(tf: TextField) extends IntegerBinding {
  bind(tf.textProperty())

  override def computeValue(): Int =
    try {
      Integer.parseInt(tf.getText())
    } catch {
      case e: NumberFormatException => 0
    }
}
```

```
class SumView extends GridPane {
  val tfInput_1 = new TextField("")
  val tfInput_2 = new TextField("")
  val tfOutput = new TextField("")

  add(tfInput_1, 0, 0)
  add(tfInput_2, 1, 0)
  add(tfOutput, 1, 1)
  add(new Label("Sum"), 0, 1)

  tfOutput.textProperty().bind(
    (new StringToIntBinding(tfInput_1)).
      add(
        new StringToIntBinding(tfInput_2)
      ).asString()
    )
}
```



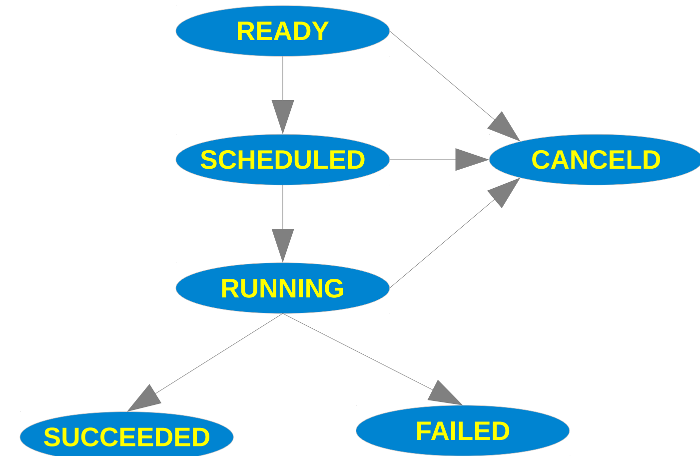
Model und Control

Worker (und ihre Unterklasse Task)

Eigenschaften als *Properties*

Ein Worker ist in einem Zustand

Der Zustand und der Fortschritt der Arbeit sind über *Properties*, d.h. sie sind beobachtbar durch andere Objekte



beobachtbare Properties:

`ReadOnlyDoubleProperty`	`progressProperty()` Gets the ReadOnlyDoubleProperty representing the progress.
`ReadOnlyBooleanProperty`	`runningProperty()` Gets the ReadOnlyBooleanProperty representing whether the Worker is running.
`ReadOnlyObjectProperty<Worker.State>`	`stateProperty()` Gets the ReadOnlyObjectProperty representing the current state.

Tasks sind beobachtbar

Property (z.B. Fortschritt) beobachten

Einfaches Beispiel: Task

- **mit Ergebnis,**
- **der nicht direkt auf GUI-Komponenten zugreift**
- **mit Ausführung in Thread-Pool (Executor)**
- **und Fortschrittsanzeige durch Fortschrittsbalken**

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

val task = new Task[Int] {
  override def call(): Int = {
    updateProgress(0, 100)  // 0 % done
    Thread.sleep(1000)
    updateProgress(20, 100) // 20 % done
    Thread.sleep(1000)
    updateProgress(40, 100) // 40 % done
    Thread.sleep(1000)
    updateProgress(60, 100) // 60 % done
    Thread.sleep(1000)
    updateProgress(80, 100) // 80 % done
    Thread.sleep(1000)
    updateProgress(100, 100) // 100 % done
    42
  }
}
```

Ein Fortschrittsbalken, dessen Anzeige an den Fortschritt des Tasks gebunden ist.

```
val progressbar = new ProgressBar
progressbar.progressProperty().bind(task.progressProperty())
```

Ergebnis des Tasks in in Textfeld anzeigen

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

Tasks sind beobachtbar

Beispiel: Fortschritt eines Tasks zur Faktorisierung beobachten

```
class ObservableFactorizingTask(nS: String) extends Task[List[Long]] {
  override def call(): List[Long] = {
    val n: Long = Integer.parseInt(nS)
    if (n < 2) throw new IllegalArgumentException()
    else {
      val result : ListBuffer[Long] = ListBuffer()
      var i = 2L
      while (i < n/2+1 && !isCancelled()) {
        if (n % i == 0 && isPrime(i)) {
          result += i
          updateMessage(s"found $i")
        }
        i = i+1
        if (isCancelled()) {
          updateMessage("Canceled")
        }
        updateProgress(i, n/2)
      }
      result.toList
    }
  }
}
```

```
val tfInput        = new TextField("")
val tfOutput       = new TextField("")
val buttonCompute  = new Button("compute")
val buttonCancel   = new Button("cancel")
val progressbar    = new ProgressBar
val tfMsg          = new TextField("")

var factorizingTask: Option[ObservableFactorizingTask] = None

buttonCompute.setOnAction(
    (ae: ActionEvent) => {
      if (factorizingTask.isDefined) {
        tfOutput.setText("I'm busy!")
      } else {
        tfOutput.setText("")
        val task = new ObservableFactorizingTask(tfInput.getText())

        progressbar.progressProperty().bind(task.progressProperty())

        tfMsg.textProperty().bind(task.messageProperty())

        ...

        executor.submit(task)
        factorizingTask = Some(task)
      }
    }
)
```

Die Weiterleitung der Ereignisse / Daten erfolgt „automatisch“ nachdem die Komponenten verknüpft wurden.