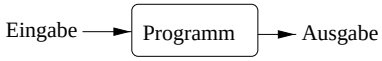
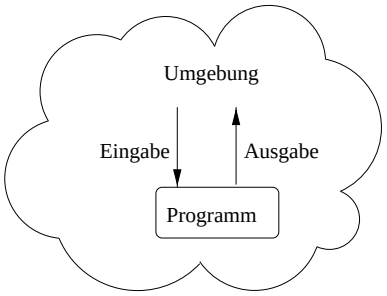
Reaktive Programmierung



Sequentiell



Reaktiv

Beispiele für reaktive Programme: GUIs, Applets, *embedded systems*

Reaktive Programme

Herkömmliches sequentielles Programmiermodell basiert auf

- Sequentielle Zustandsübergänge
- Ein/Ausgabe durch Seiteneffekte

Hier unzureichend:

- Grundlegende Operationen: auf Eingabe warten, Ausgabe senden
- Mehrere Eingaben gleichzeitig, Reihenfolge der Ausgabe nicht spezifiert, Nebenläufigkeit

Problem: Realisierung reaktiver Programme in sequentieller Sprache

Realisierung reaktiver Programme

- Eventschleife im Program (X Windows)
- Externe Eventschleife:
  - Callbacks (Tcl/Tk)
  - Signale (Qt, GTK+)
  - Event handlers (Java: Applets, Swing, AWT)
- Events as first-class values
- Kontinuierliches Verhalten

Zentrale Eventschleife: Xlib

```
#include <X11/Xlib.h>                       #include <X11/Xutil.h>
int main(int argc, char **argv)
{Display *dply; XEvent event;
 /* draw some graphics here */
 for (;;) {
     XNextEvent(dply, &event);
     if (event.type == ButtonPress) {
         printf("Button pressed at (%d, %d)\n",
                event.xbutton.x, event.xbutton.y);
       } else if (event.type == KeyPress) {
         KeySym ks;
         ks= XLookupKeysym(&event.xkey, 0);
         printf("Key %04x (%s) pressed.\n", ks, XKeysymToString(ks));
       }
     XFlush(dply);
   }}
```

Zentrale Eventschleife

Vorteile:

- Direkte Kontrolle
- Schnelligkeit

Nachteile:

- umständlich, fehleranfällig
- Nicht kompositional

Eventbehandlung mit Callbacks, Signals und Event handlern

Eine Eventschleife *außerhalb* des reaktiven Programms

Programm implementiert Prozeduren, die Events verarbeiten.

Prozedurparameter: Event-Daten, zum Beispiel gedrückte Taste oder Mausknopf; Zusatzinformationen wie Mausposition.

Für Beispiel oben wären das:

```
typedef struct {int button; int x_pos; int y_pos} mouse_info;

void handle_buttonPress(mouse_info i);
void handle_keyPress(char key);
```

Unterschied zwischen Tcl/Tk, Qt/GTK+, AWT/Swing: Art der Bindungsbehandlung

Callbacks in Tcl/Tk

- Tcl/Tk: *user interface library and programming language*
- Tcl (tool command language): String-basierte Kommandosprache
- Tk (graphical toolkit): Bibliothek von Funktionen zum Erstellen von GUIs
- Philosophie: Leichtgewichtige Skriptsprache, schnelle Entwicklung von GUIs durch das Zusammenkleben von Applikationen (*glueware*)
- Plattformunabhängig: X, Windows, MacOS, ...

Callbacks in Tcl/Tk

- Bindung von Kommandos an Events durch `bind`:

```
button .b -text Test
bind .b <Enter> {puts "Button Entered!"}
bind .b <ButtonPress-2> { puts "Button pressed at (%x, %y)" }
pack .b
```

- Völlig ungetypt
  Funktionen höherer Ordnung durch *Quoting*
  Implizite Parameterübergaber über %-Substitutionsmechanismus
- Ablauf:

  Aufbau des Interface
  Festlegung des dynamischen Verhaltens
  Interface darstellen und externe Eventschleife aufrufen

Signale in GTK+

- GTK+: GUI-Bibliothek für Linux (GNOME)
- Im Prinzip plattformunabhängig (hauptsächlich X/Unix)
- Implementiert in C, aber Bindungen an andere Sprachen (Python, Scheme, C++, Pascal, Eiffel, Haskell) verfügbar.
- Externe Ereignisse generieren *Signale*, an die Funktionen gebunden werden
- Qt (KDE): ähnlich, aber C++

Beispiel

```
#include <gtk/gtk.h>
void click_button_handler(GtkWidget *w, gpointer gdata)
{ g_print("Button clicked\n"); }

int main(int argc, char argv[])
{ GtkWidget *button;
  ... /* Initialisierung, Fenster kreiieren */
  button= gtk_button_with_new_label("Click me!");
  gtk_signal_connect(GTK_OBJECT(button), "clicked",
                     GTK_SIGNAL_FUNC(click_button_handler), NULL);
  gtk_widget_show(button);
  ... /* Button in Fenster packen, Fenster anzeigen */
  gtk_main();
}
```


Event Handling in Java

Eventverabeitende Methoden leben in Unterklassen der `Listener`-Klassen:
- `interface ActionListener` für Aktionen (z.B. Knopfdrücke);
- `interface MouseListener` für Mouse-Events.

In `Listener` werden Event-bearbeitende Methoden überladen:

```
interface ActionListener {
 public void actionPerformed(ActionEvent e)
}
```

Mit `addActionListener(Object o)`, `addMouseListener(Object o)` werden Listener an Event-Quellen gebunden.

Beispiel: ein einfaches Applet

```
public class Beeper extends JApplet
                    implements ActionListener {
    JButton button;

    public void init() {
        button = new JButton("Click Me");
        getContentPane().add(button, BorderLayout.CENTER);
        button.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        Toolkit.getDefaultToolkit().beep();
    }
}
```

Zusammenfassung: Callbacks, Signals und Event Handlers

Vorteile:

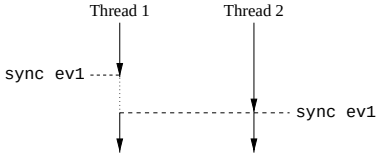
- abstrakter als explizite Eventschleife
- kompositional innerhalb des Toolkits (e.g. zwei Swing-Programme)

Nachteile:

- Mangelnde Modularität und Kompositionalität, weil es immer nur eine Eventschleife geben kann
- Zustandsbasiert

Funktionale reaktive Programmierung: `Events` für Haskell

Eine grundlegende Operation: *Synchronisation*



Beschreibung der Synchronisationspunkte: `Event`

Kombination von Events mit `+>`: Auswahl unter mehreren Möglichkeiten

```
type Event a
sync   :: Event a -> IO a
(+>)   :: Event a -> Event a -> Event a
(>>>=) :: Event a -> (a-> IO b) -> Event b
```


Abgeleitete Events

Mit `always :: Event a` kann immer synchronisiert werden.

Events als Monaden:

```
instance Monad Event where
  (>>=) ev1 getEv2 = ev1 >>>= sync . getEv2
  return v         = always (return v)

-- Damit z.B:
forever :: Event a-> Event ()
forever e = do e; forever e

spawnEvent :: Event a-> IO (IO ())
spawnEvent r = do i<- forkIO (sync r); return (killThread i)
```


Beispiel: Events in `HTk`

```
main :: IO ()
main =
  do htk <- initHTk [text "Main Window"]

     but1 <- newButton htk [text " Quit ", size(10, 2)]
     pack but1 []

     (press, _) <- bindSimple but1 (ButtonPress (Just (BNo 1)))

     spawnEvent (forever (press >>>= (putStrLn "Exiting!" >> destroy htk)))

     (htk_destr, _) <- bindSimple htk Destroy
     sync (htk_destr)
```

HTk

HTk: graphische Benutzerschnittstellen für Haskell

Eigenentwicklung Uni Bremen

Benutzt Tk, aber ersetzt Tcl durch Haskell

Voll nebenläufig

Getypten, komponierbaren Events als „first-class objects“

Deckt den gesamten Funktionsumfang von Tk ab:
widgets, *text widgets*, *canvasses*, ...

Ein weiterer Ansatz: Kontinuierliches Verhalten

Anderer Ansatz: `Verhalten` als Basistyp: FAL, FranTk (Hudak, Sage)

```
type Time = Float
type Behavior a = Time->
```

Events sind diskrete Ereignisse:

```
type Event b = (Time, b)
```

Events können das Verhalten ändern und andersherum:

```
untilB :: Behavior a -> Event (Behavior a) -> Behavior a
when, while :: Behavior Bool -> Event ()
```


Kontinuerliches Verhalten in FAL und FranTk

Veränderliche Events:

```
(=>>) :: Event a -> (a->b) -> Event b
snapshot :: Event a -> Behavior b -> Event (a,b)
```

Basisevents z.B. `lbp :: Event ()`, `key :: Event Char`

Modellierung des sichtbare Verhalten des Programmes:

- `Behavior String` — kontinuierliche textuelle Ausgabe
- `Behavior Graphic` — Grafische Ausgabe
- `Behaviour Float` — hybride Ausgabe, z.B. Kontrolle eines Druckventils

Analytische Funktionen, z. B. `integral :: Behavior Float -> Behavior Float`

Ein Beispiel für FAL

```
ball :: Behavior Picture
ball = paint red (translate (x,y) (ell 0.2 0.2))
       where g = -4
             x = -3 + integral 0.5
             y = 1.5 + integral v
             v = integral g `switch` (hit `snapshot` v =>> \(_, w)->
                                      lift0 (-w) + integral g)
             hit = when (y <* -1.5)
```

wobei `a 'switch' b = (a `untilB` b) `switch` b`

Animierung des Beispiels oben durch

```
reactimateP:: Behaviour Picture-> IO ()

main :: IO ()
main = reactimateP "Bouncy Ball" ball
```

Hautproblem: wieder eine zentrale Event-Schleife (wenn auch sehr abstrakt)

- Reaktive Programme mit traditionellen sequentiellen Programmiermodell schwer zu fassen — Ansätze:

  Interne Eventschleifen (Programm durchläuft alle Eingaben)

  Externe Eventschleifen (Programm als Sammlung von Funktionen, die Eingaben behandeln)

  Nachteile: Kompositionalität und Modularität

- Funktionale Sprachen erlauben abstraktere Ansätze:

  Kontinuierliches Verhalten (FAL/FranTk)

  Ereignisbasierte Programmierung (HTk)

  Vorteile HTk: kompositional und modular, nebenläufig