Graphische Benuterschnittstellen (GUIs)

OpalWin

Peter Pepper
Christoph Höger
Robert Reicherdt
Martin Zuber
Tutoren

Prinzipien von GUIs

In den meisten GUI-Systemen gibt es ...

- vordefinierte Komponenten mit impliziten Zeichenroutinen (`Label, Button, ...`)
- vom Benutzer selbst programmierte Zeichnungen (`Canvas, paint, ...`)

Bei den Komponenten unterscheidet man oft

- atomare Komponenten (Label, Button, ...)
- Container (Komponenten, die andere Komponenten enthalten)

Prinzipien von GUIs

In den meisten GUI-Systemen gibt es ...

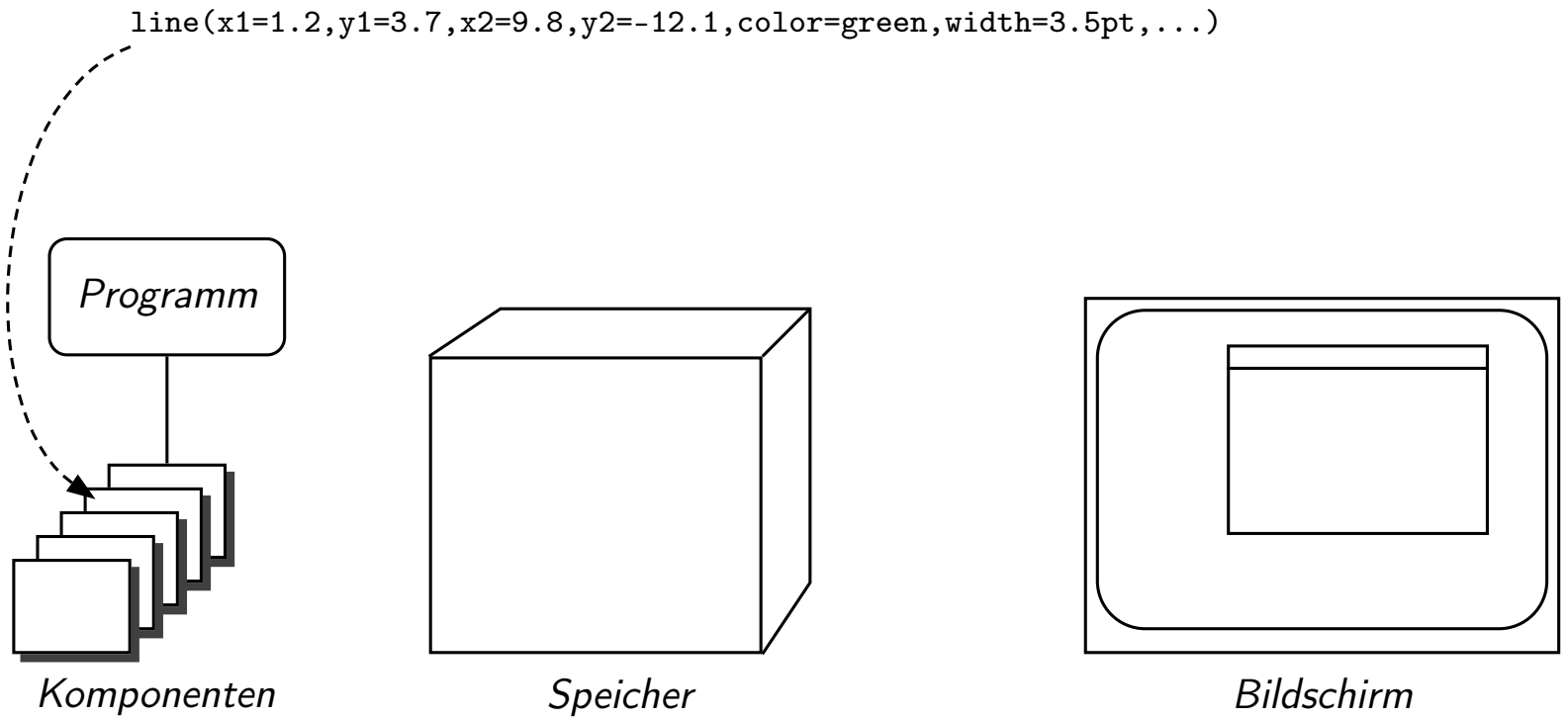
- vordefinierte Komponenten mit impliziten Zeichenroutinen (`Label, Button, ...`)
- vom Benutzer selbst programmierte Zeichnungen (`Canvas, paint, ...`)

Bei den Komponenten unterscheidet man oft

- atomare Komponenten (Label, Button, ...)
- Container (Komponenten, die andere Komponenten enthalten)

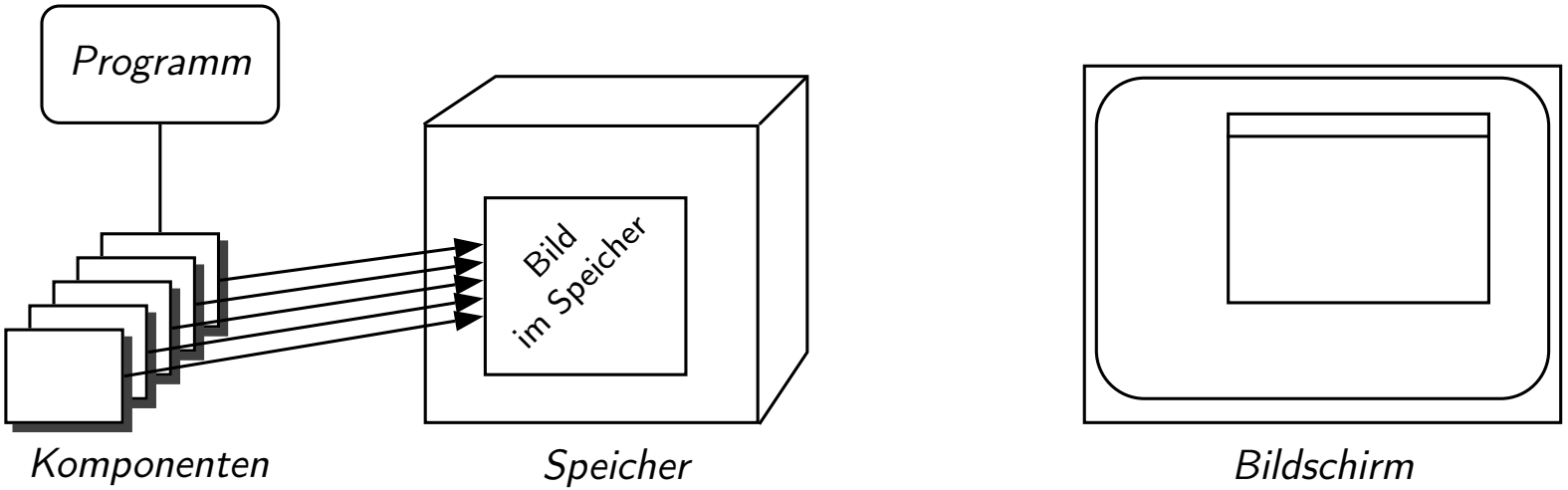
Vom Programm zum Bildschirm-Fenster

Im Program stehen nur codierte Beschreibungen von Bildern



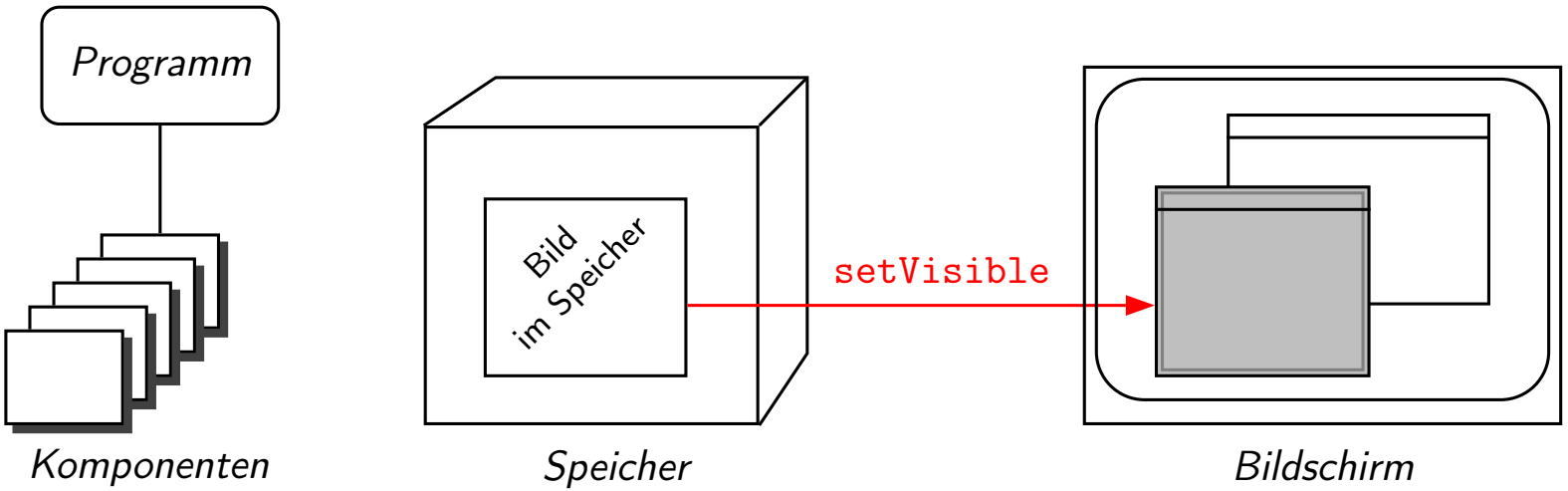
Vom Programm zum Bildschirm-Fenster

Aus diesen Beschreibungen wird im Speicher eine Pixelmatrix erzeugt. (Das erledigen GUI-Systeme automatisch.)



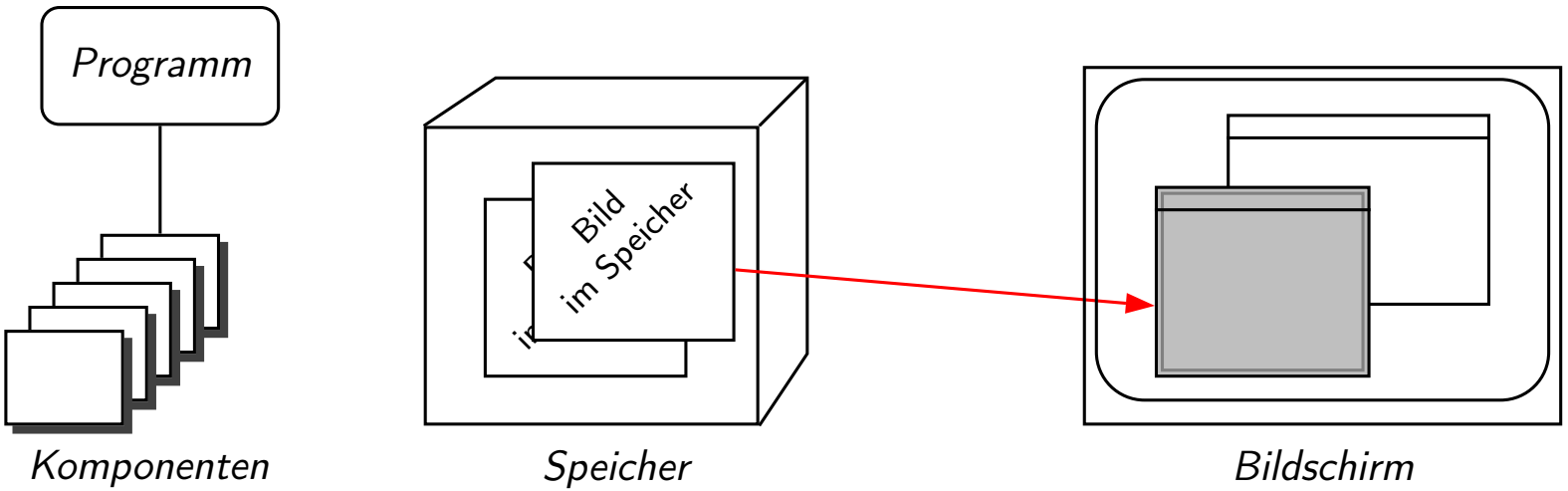
Vom Programm zum Bildschirm-Fenster

Das Bild wird sichtbar durch die Übertragung der Pixelmatrix auf den Bildschirm.



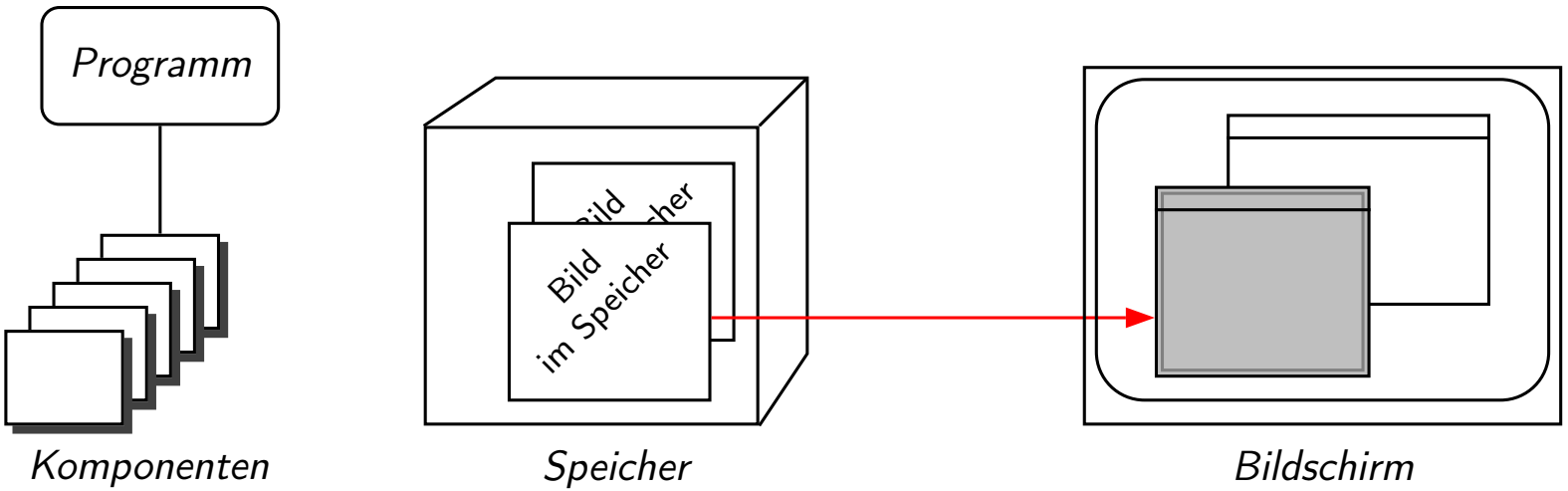
Vom Programm zum Bildschirm-Fenster

Doppelpufferung optimiert das Erscheinungsbild.

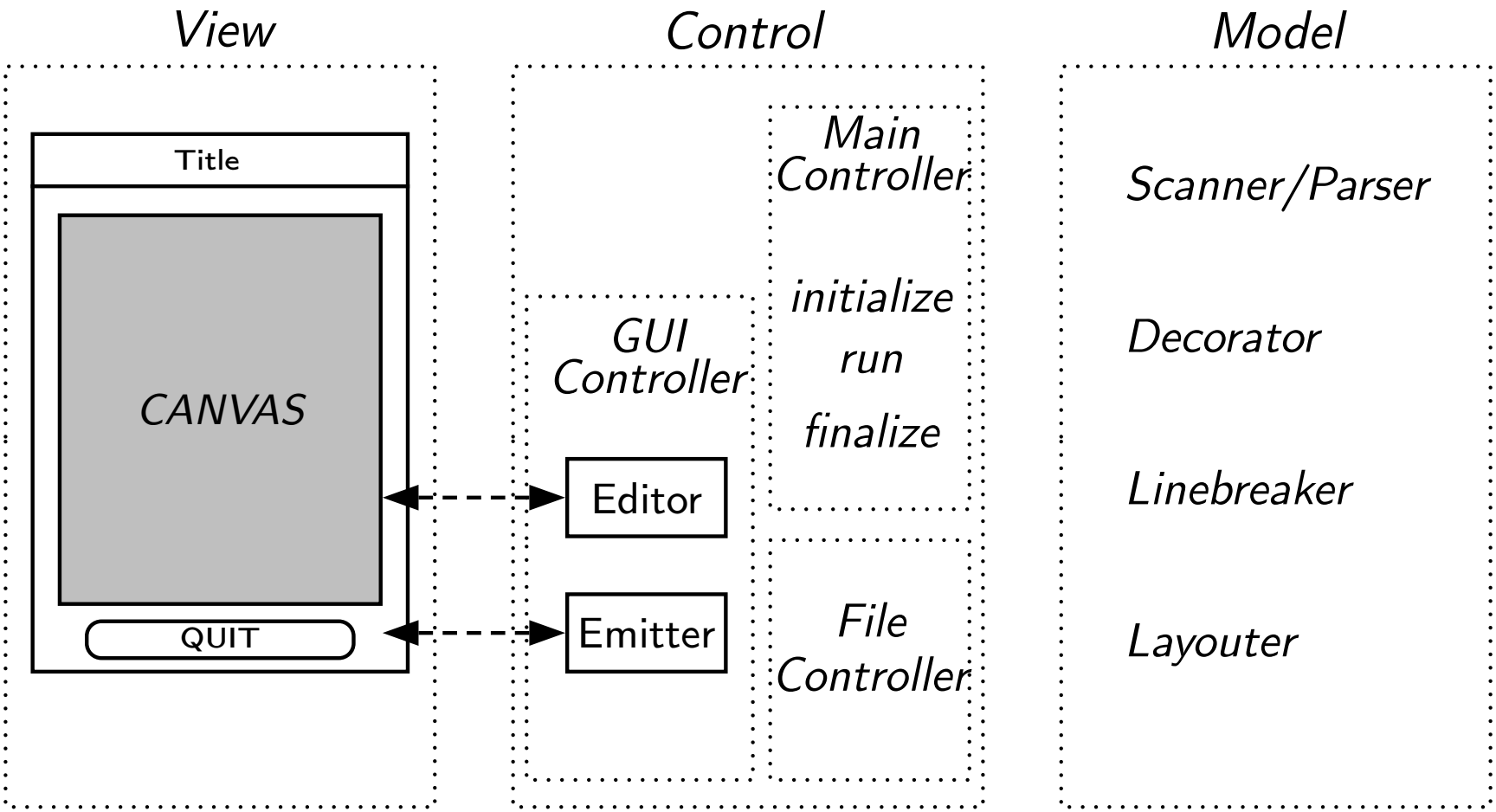


Vom Programm zum Bildschirm-Fenster

Doppelpufferung optimiert das Erscheinungsbild.



Model–View–Control



Globale Programmstruktur

FUN main: com[void]

```
DEF main == initialize
            & run
            & finalize
```

```
DEF main == initialize      & λinfo.
            run(info)       &
            finalize(info)
```

FUN intialize: com[info]
FUN run: info → com[info]
FUN finalize: info → com[void]

GUIs – Model-View-Control

Initialisierung

- Beschaffung der Programmparameter
- Beschaffung der relevanten Systemdaten
- Einlesen von Dateien

FUN intialize: com[info]

```
DEF initialize == getFileName    & λ name.
                  initializeGui  & λ guiInfo.
                  readFile(name) & λ fileInfo.
                  yield(makeInfo(fileInfo, guiInfo))
```

Hinweis: Der Opal-Compiler erwartet i.Allg. mehr Klammern als hier angegeben sind!

Sammlung der Initialisierungsdaten

SIGNATURE `Info`

```
TYPE info = makeInfo( file: fileInfo,
                      gui: guiInfo    )
TYPE fileInfo = makeFileInfo( name: denotation,
                              content: string   )
TYPE guiInfo = makeGuiInfo( title: denotation,
                            metrics: metrics,
                            ...                 )
```

Wenn Programme viele Attribute haben, ist es sinnvoll, diese strukturiert in einigen wenigen (Tupel-)Typen zu sammeln. Dies verbessert die Struktur und Lesbarkeit der Programme erheblich.

Umfangreiche Typen (wie z.B. `guiInfo`) können auch in eigenen Strukturen definiert und hier importiert werden.

Hilfsstruktur Error für die Fehlerbehandlung

Die Fehlerbehandlung sollte nach einem einheitlichen Schema erfolgen. Deshalb fasst man alle Funktionen zur Fehlerbehandlung in einer Struktur zusammen.

(Bei sehr vielen Fehlerarten kann man die Fehlerfunktionen auch in mehrere Strukturen aufteilen.)

SIGNATURE `Error`

```
FUN argError: com[void]
FUN fileError: fileInfo → com[fileInfo]
    ...
```

Diese Struktur muss dann überall importiert werden, wo Errorhandling gebraucht wird.

Initialisierung: Zugriff auf die Kommandozeile

FUN getFileName: com[denotation]

```
DEF getFileName == argCount & λ anz.
                   IF anz = 2 THEN arg(1)
                              ELSE argError & exit(1) FI
```

Import aus BibOpalica:

```
SIGNATURE ProcessArgs
  FUN argCount: com[nat]
  FUN arg: nat → com[denotation]
  FUN args: com[seq[denotation]]
      ...
```

Initialisierung: Einlesen der Datei

FUN readFile: denotation → com[fileInfo]

```
DEF readFile(name) == open(name, "r")        & λ file.
                      read(file, max'Nat)    & λ content.
                      close(file)            &
                      yield(makeFileInfo(name, content))
                      |
                      fileError(makeFileInfo(name, ◊))
```

Import aus BibOpalica:

```
SIGNATURE File
  SORT file
  FUN open: denotation × denotation → com[file]
  FUN close: file → com[void]
  FUN read: file × nat → com[string]
      ...
```

Initialisierung: Beschaffung der GUI-Elemente

Die GUI-Elemente stammen aus mehreren Quellen:

- Attribute, die der Programmierer vorgibt (als Konstanten):
  (Titel, Farben, Fonts, ...)
- Attribute, die vom Windows-System abgefragt werden müssen:
  (Bildschirm-Auflösung, -höhe und -breite, Font-Metrik, ...)
- Schnittstellen zwischen graphischem Layout und Programm:
  (Emitter, Regulator, Editor, ...)

Diese werden am besten in einer gemeinsamen Struktur definiert, die sowohl die einschlägigen Konstanten enthält als auch die Kommandos zur Beschaffung der System-Atrribute.

GUIs – Model-View-Control

Gui-Initialisierung: Sichtbare Schnittstelle

SIGNATURE `GuiInfo`

```
TYPE guiInfo = makeGuiInfo(
       title: denotation,          // Fenstertitel
       anchor: point,              // links oben
       size: dimension,            // Breite/Höhe
       font: font,                 // Schriftart
       fg: color,                  // Vordergrund-Farbe (Schrift)
       bg: color,                  // Hintergrund-Farbe
       ...
       metrics: metrics,           // Font-Metrik
       paint: canvasEditor,        // Textschnittstelle (gate)
       quit: emitter               // Buttonschnittstelle (gate)
...
FUN initializeGui: com[guiInfo]
```

GUIs – Model-View-Control

Gui-Initialisierung: Versteckte Definitionen

IMPLEMENTATION GuiInfo

```
FUN title: denotation
FUN font: font
FUN bg: color
    ...
DEF title == "HTML – Viewer"
DEF font == variable'WinConfig
DEF bg == lightgrey'WinConfig
    ...
DEF intializeGui ==
    emitter              & λ quit.        // gate for QUIT button
    canvasEditor         & λ editor.      // gate for text-drawing area
    fontMetrics(font) & λ metrics.
    yield(makeGuiInfo(title, font, ..., metrics, editor, quit))
```

GUIs – Model-View-Control

. . . und noch eine hübsche Hilfsfunktion

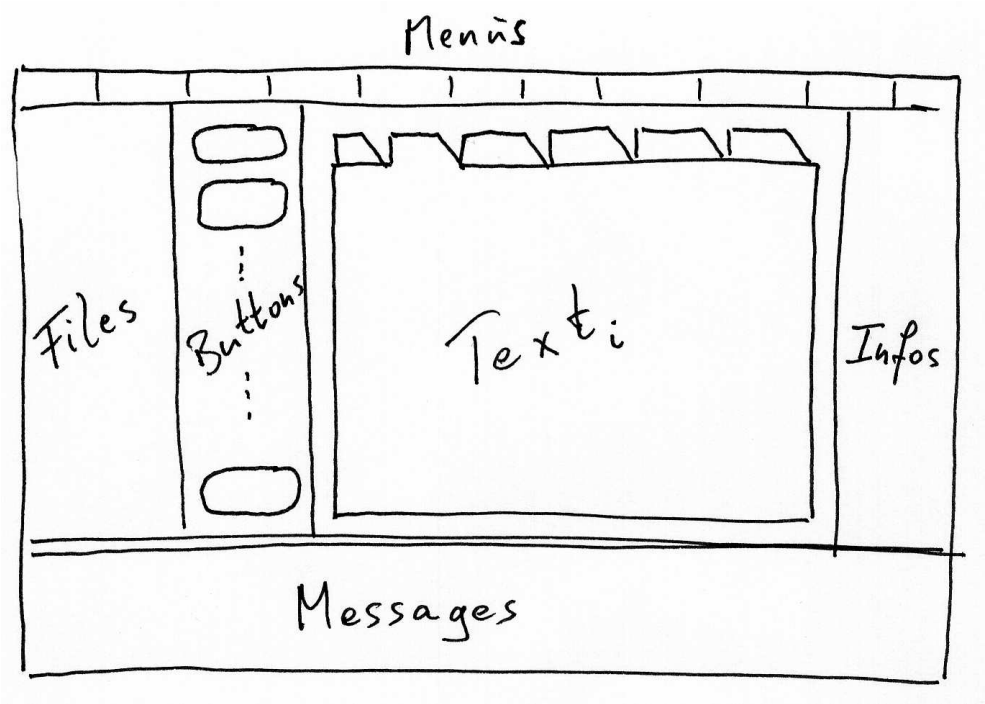
IMPLEMENTATION `Dot`$[\alpha, \beta]$

SORT α β
FUN . : $\alpha \times (\alpha \to \beta) \to \beta$
DEF `x.f` $==$ `f(x)`

$\rightsquigarrow$ Notationen analog zu objektorientierten Sprachen (Java, . . .)
z.B. `fileInfo.content`, `guiInfo.font`, `guiInfo.metrics`, `...`

GUIs – Design

Zuerst kommt die Skizze . . .



SplitPane
BorderLayout
BoxLayout
TabbedPane

MenuBar, Menu, MenuItem
Button
TextArea
FileList
...

GUIs – Design

Layout-Elemente + Controller-Gates