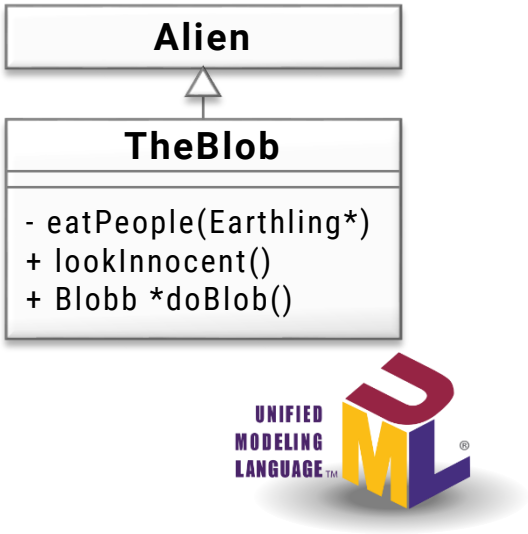
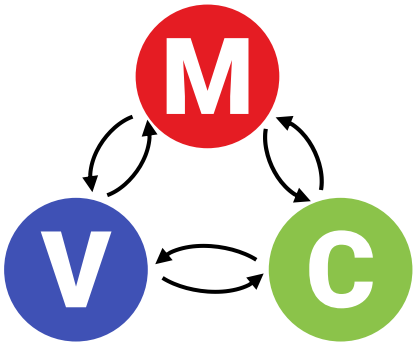
Einführung in die Softwareentwicklung

SOMMERSEMESTER 2020

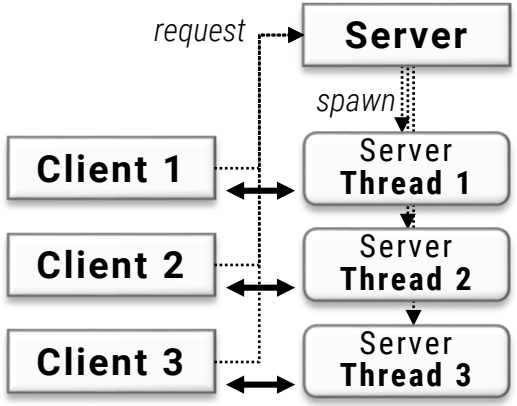
Institut für Informatik

JG|U JOHANNES GUTENBERG UNIVERSITÄT MAINZ





Design Patterns



Architectural Patterns

Foliensatz #13

Design & Muster II: GUIs

Michael Wand · Institut für Informatik · michael.wand@uni-mainz.de

Graphische Benutzerschnittstellen
(OOP + „Events“)

Oberklasse: Widgets

Oberklasse:
Ein Widget ist ein rechteckiges GUI-Element.

Widget.h

```
class Widget {

 public:
   int x, y, width, height; // geometry

   // protected: Interface für geerbte Klassen
   // Idee: „Event-oriented programming“
   // „Hollywood architecture“ – we call you!

   // called to draw the widget (e.g., if window appears)
 protected:
   virtual void draw(Graphics *g) = 0;

   // called on user events (Mouse, keyboard)
   virtual void mouseDown(int x, int y, int buttonNo) = 0;
   virtual void mouseMove(int x, int y) = 0;
   virtual void mouseUp() = 0;

   virtual void keyPressed(char key) = 0;
};
```

Oberklasse: Widgets

Oberklasse:
Ein Widget ist ein rechteckiges GUI-Element.

Widget
+ x, y, width, height: unsigned
*# draw(Graphics *g)* *# mouseDown(int x, int y, int buttonNo);* *# mouseMove(int x, int y)* *# mouseUp()* *# keyPressed(char key)*

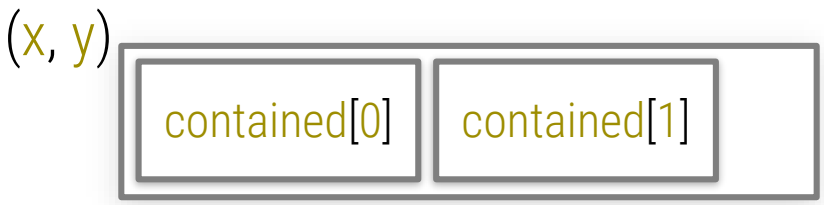
Gui-Framework (schematisch)

Label ist ein beschriftetes GUI-Element.

Label.h (cpp-Code integriert)

```
class Label : public Widget {
 public:
   std::string text;

   // draws text in widget area (schematic!)
   virtual void draw(Graphics *g) {

      g.drawText(x, y, text);

   }

 protected:   // Labels do not react to user input

   virtual void mouseDown(int x, int y, int buttonNo) {}
   virtual void mouseMove(int x, int y) {}
   virtual void mouseUp() {}

   virtual void keyPressed(char key) {}
};
```

Baum von Widgets



Samlung von Widgets

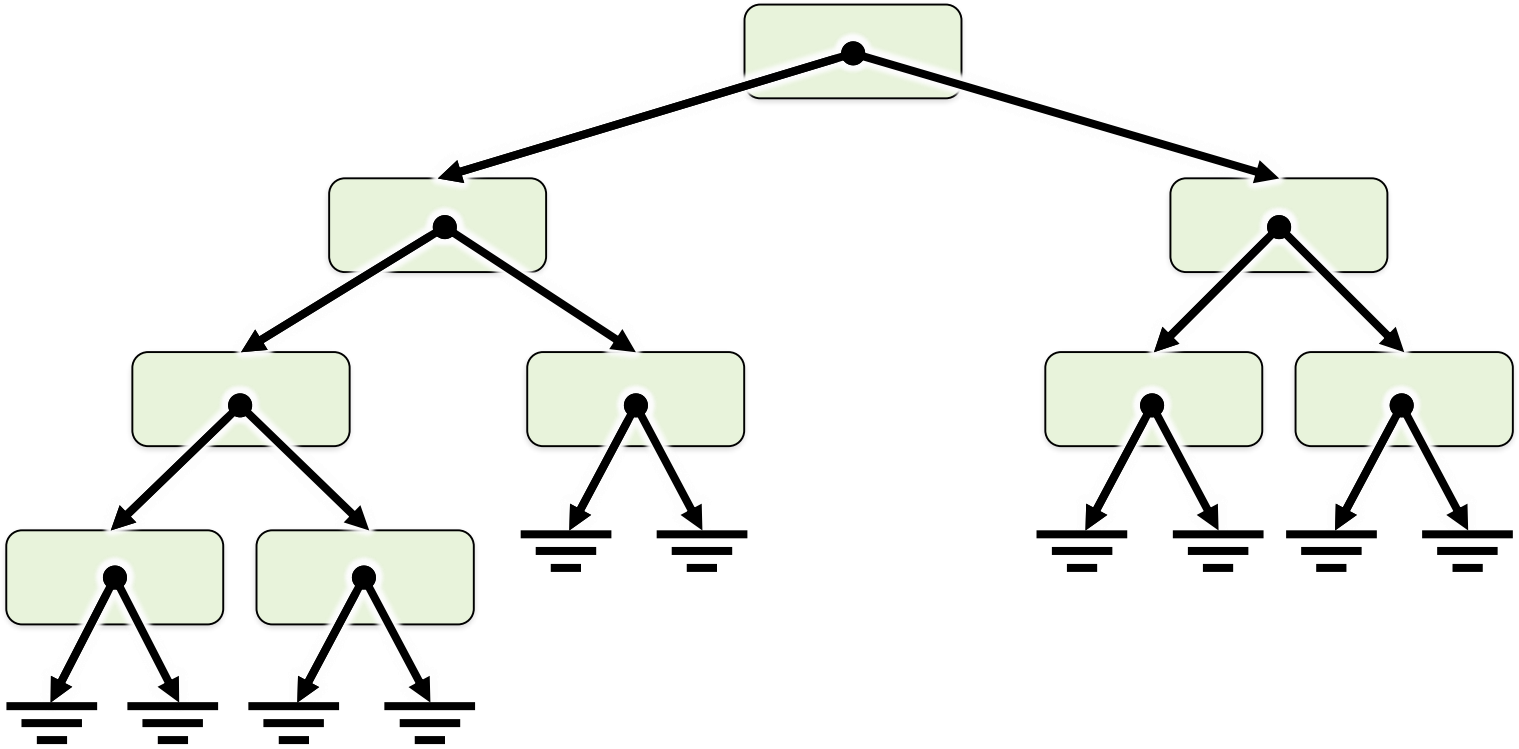
HBoxList: horizontale Anordnung

z.B. eine Toolbar

Klasse: HBoxList

```
class HBoxList : public Widget {
 private:
   std::vector<Widget*> childWidgets; // contained widgets
 protected:
   virtual void draw(Graphics *g) { /* call draw on all contained widgets */ }
   // called on user events (Mouse, keyboard)
   virtual void mouseDown(int x, int y, int buttonNo)
      { /* check which widget was hit, then call "mouseDown" */ }
   virtual void mouseMove(int x, int y)
      { /* check which widget was hit, then call "mouseMove" */ }
   virtual void mouseUp()
      { /* check which widget was hit, then call "mouseUp" */ }
   virtual void keyPressed(char key);
      { /* check which widget has keyboard focus, then call its "keyPressed" */ }
};
```

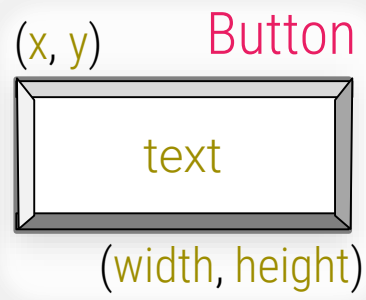
Widget-Hierarchie: Objektbaum



Gui-Frameworks (schematisch)

Klasse „Button“

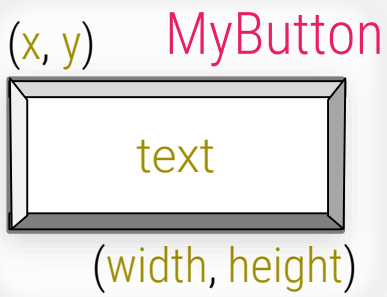
```
class Button : public Label {

 protected:
   // override in subclass to define what should happen!
   virtual void buttonPressed() = 0;

   // draws button around text
   virtual void draw(Graphics g) {

      // draw some 3D-shaded button (fictious graphics library)
      GfxUtils.drawButton(g, x, y, width, height);

      // inherited – draw text on top of button (Button extends Label)
      Label::draw(g);

   }

   virtual void mouseDown(int x, int y, int buttonNo) {

      if (buttonNo == 0) { // left mouse button

         buttonPressed();

      }

   }

};
```



Nutzung der „Button“ Klasse

Klasse „MyButton“

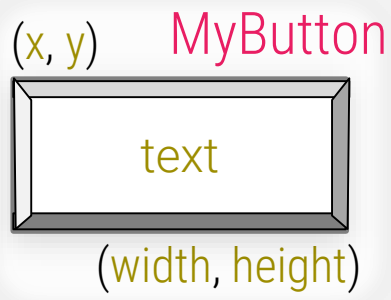
```
// This defined by the user of the GUI Framework!
class MyButton : public Button {

   // override in subclass to define what should happen!
   protected void buttonPressed() {

      cout << "Button was pressed!";

   }

};
```



Benutzung der Button Klasse

Klasse „MyButton“

```
// This defined by the user of the GUI Framework!
class MyButton : public Button {

   ApplicationObject *appObjReference;

   // override in subclass to define what should happen!
   virtual void buttonPressed() {

      // forward the event to my own application...
      appObjReference->performSomeAction();

   }

};


// later, somewhere in the application code...
Button *b = new MyButton();

b->x = ...; b->y = ...; b->width = ...; b->height = ...;
b->text = "Start Server";
b->appObjReference = myApplication->somePart->serverStartup;

HBoxList *toolbar = ...; // retrieve application toolbar, build earlier
toolbar->add(b);
```



Eventbehandlung

Reale Implementierung?

- JAVA AWT (abstract window toolkit)
  - Design sehr ähnlich zu unserem Beispielcode
  - Zu einfach für komplexe System
  - Immer noch verfügbar (Teil von JAVA SWING)

Die Sache ist etwas umständlich!

- Eigene Unterklasse für jedes eigene Objekt
- Bessere Event-behandlung nötig

Event-Queues/Loops

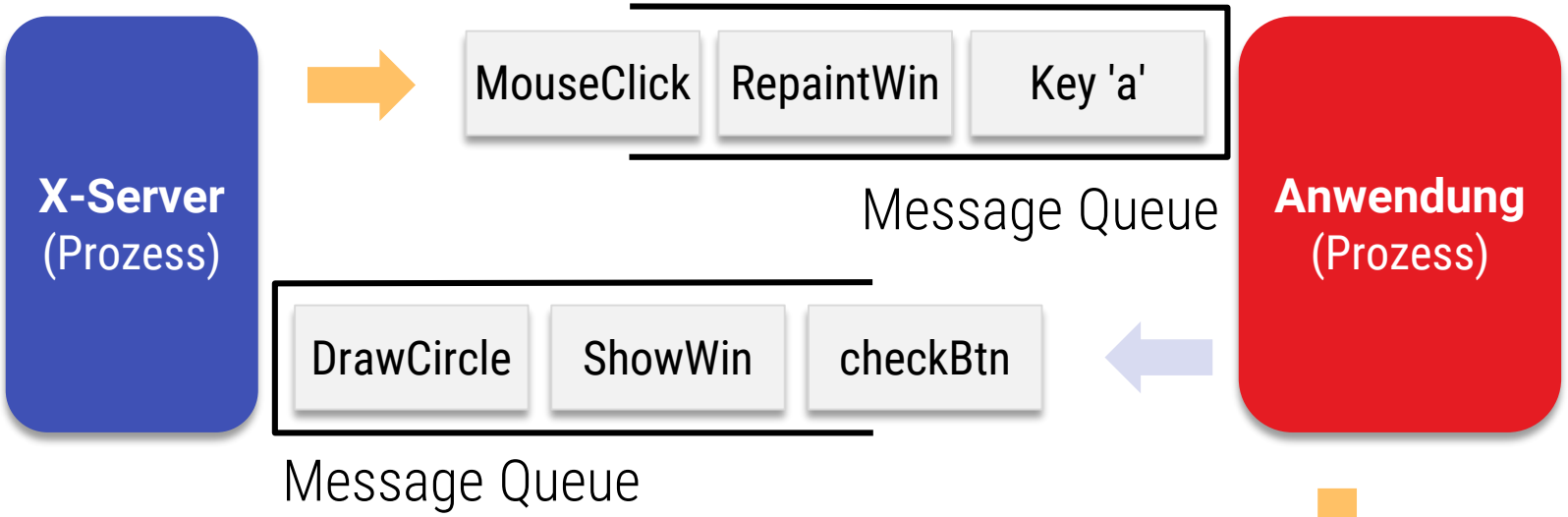
(Ereignisorientierte Architektur)

Ereignisorientierte Programmierung

Ereignisorientierte Programmierung

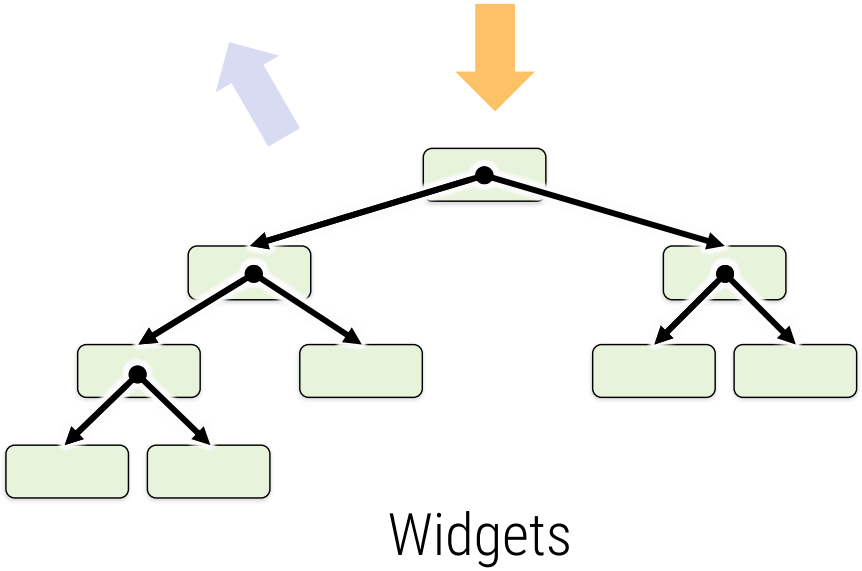
- Methodenaufruf, wenn Ereignis eingetreten ist
- Hollywood-Prinzip

Ereignisorientierte Programmierung

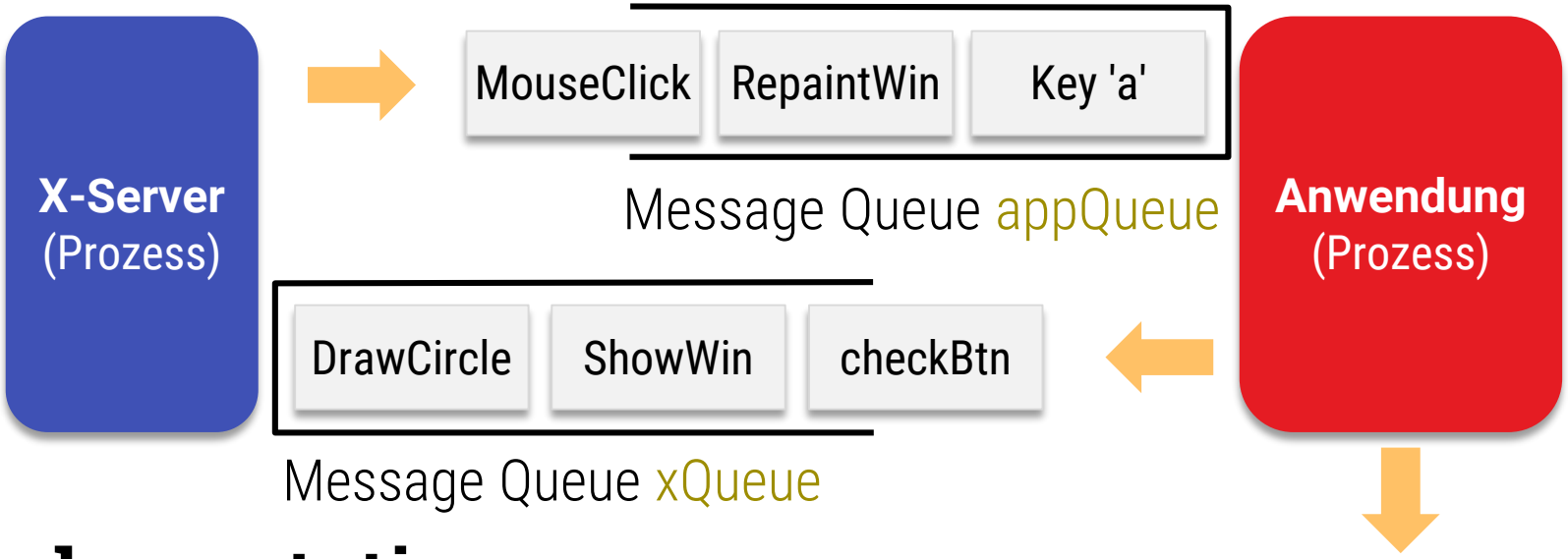


Implementation

- Messages & Callbacks
- Fenstermanager als nebenläufiger Prozess



Ereignisorientierte Programmierung



Implementation

- Messages & Callbacks
- Fenstermanager als nebenläufiger Prozess
- „Eventloop“ erzeugt Ereignisse

```
void eventLoop(appQueue, topWindow) {
  while (true) {
    msg = waitMsg(appQueue);
    if (msg.type=mouseDown) {
      topWindow.mouseDown(msg.x,msg.y);
    } else if (msg.type=mouseUp) {
      topWindow.mouseUp(msg.x,msg.y)
    } else if {
    …
  }
}
```

Pseudo-Code!

Design Patterns

Observer Pattern

- Widgets als Observer
- Passives Verhalten
- „Reactive“

Vorteil

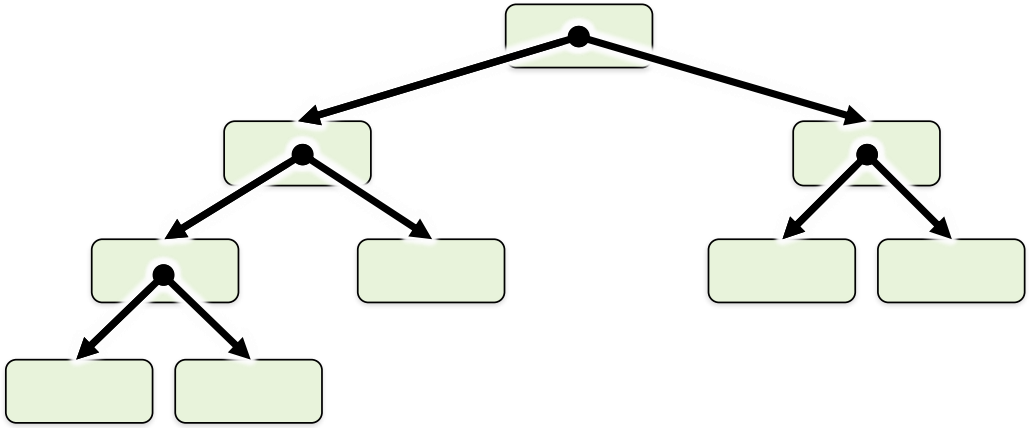
- Erweiterbarkeit, Viele Widgets operieren parallel
- Widgets hinzufügen ohne Ereignisvert. zu ändern

Nachteil

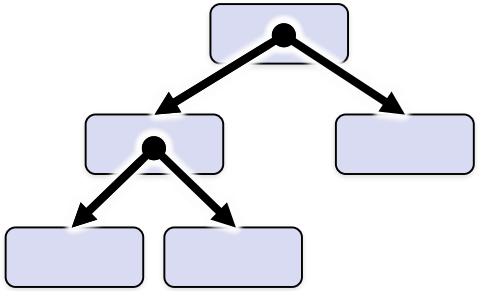
- Unintuitiv, Abläufe werden aufgebrochen

Ereignisse im Widget-Baum

Event-Behandlung

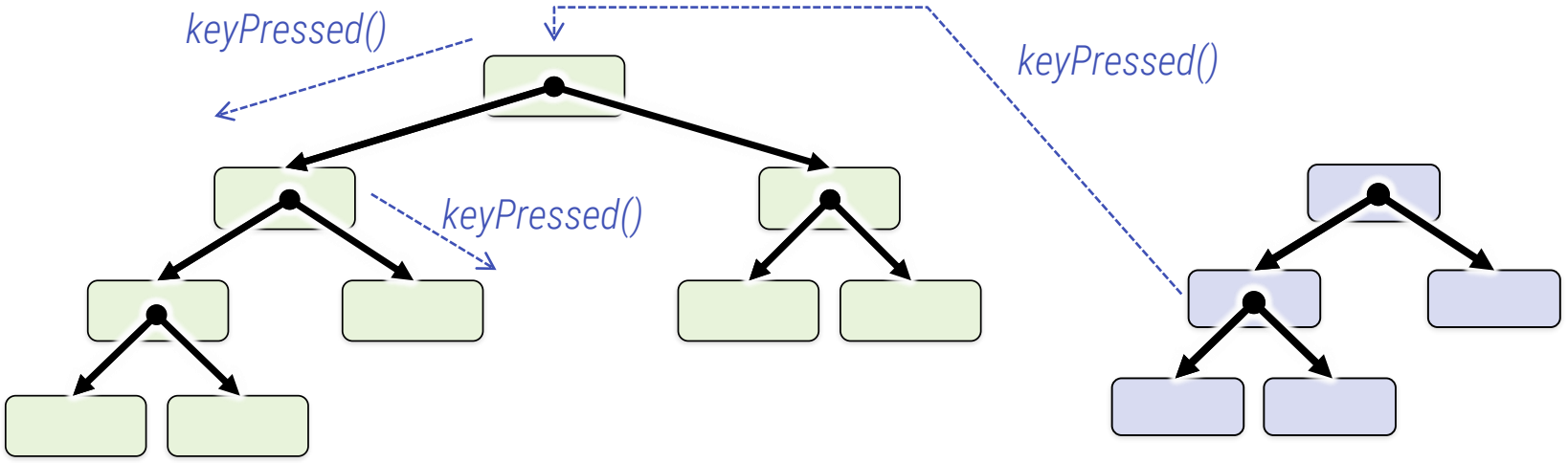


Widget Hierarchie



Anwendungs-
objekte

Event-Behandlung



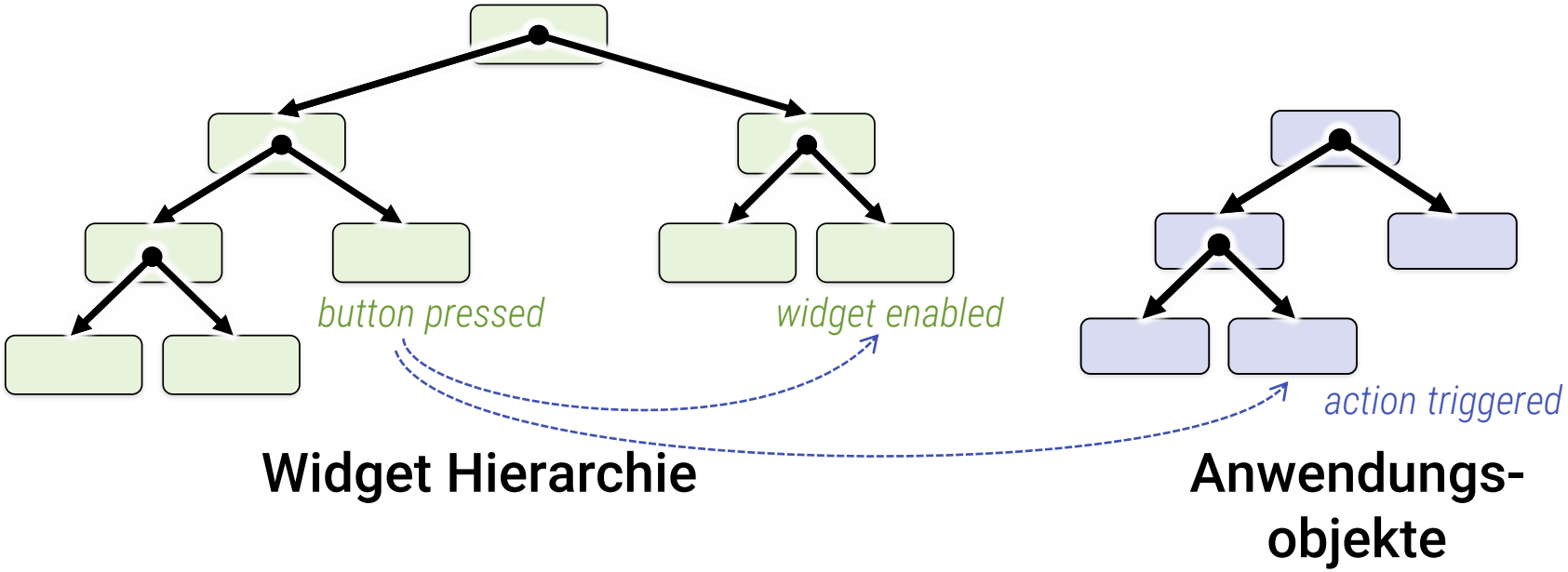
Widget Hierarchie

Anwendungs-objekte

Hierarchische Event-Verteilung

- Über Membermethoden gut möglich
- Anwendung oder System schickt Event an Hauptfenster, hiearchische Verteilung

Event-Behandlung



Querverweise

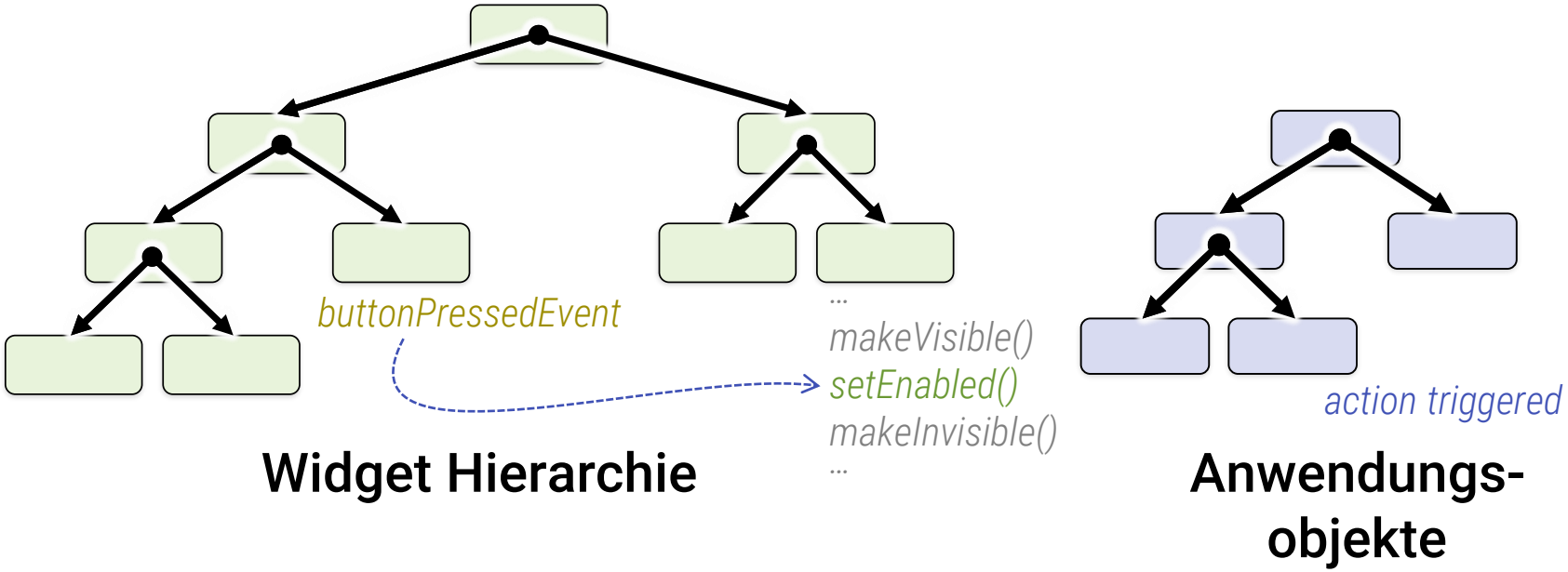
- Neuer Mechanismus nötig
- Nachrichten quer zur Hierarchie versenden

Lösungen

Mögliche Lösungen

- Java-AWT, EIS-Toy-Beispiel
  Abstrakte Methoden für Subklassen
  - Typischerweise: Nachfahre enthält Zeiger auf Empfänger
- Delphi
  Delegation: Zeiger auf Membermethoden von Objekten
  - Zeiger auf ein Objekt und eine seiner Methoden
    („Pointer to `function of object`“)
- Qt, C#
  Delegates / Signales+Slots:
  - Listen von Empfängern möglich
  - Sicher (keine „dangeling references“)

Event-Behandlung



Querverweise

- Event-Variablen an Widgets (in Qt: Listen)
- Empfänger zuordnet (Objekt, Methode)

Beispiel: QT

Abstrakt gesehen…

Qt-Architektur

- Objekthierarchie:
  - Owner-basiert
  - Strikt Baumförmig (keine Querverweise außer Events)
- Infrastruktur:
  - Introspection/Reflection via MOC
    - Serialisierung
    - GUI-Builder via Reflection
    - Propertysystem (setter-getter)
    - Versand von Objekten zwischen Prozessen

Abstrakt gesehen…

Qt-Architektur

- Eventhandling
  - Hierarchische Events (Klassisch)
  - Signals + Slots für Querverweise
    - Buchführung vermeidet „dangeling References“
- Spracherweiterung
  - Definiere `signals`, `slots`
  - „`emit signal`“, „`connect signal+slot`“
- „Framework“
  - public-Interface für Instanzen
  - protected-Interface für Nachfahren (viel vordefiniert)
  - private-Interface kann sich ändern

QT Signals + Slots

```
#include <QtCore/qobject>

// Jedes QObject kann Signale empfangen
class MyClass : public QObject {

   Q_OBJECT  // Präproz. Makro, setzt Code für Refleciton ein (moc.exe nutzen)

 signals:

     void anEvent();
     void twoNumbers(int, int);

 public slots:

     void crashApplication() { ((void*)0) = 42; /* hooray!! */ }
     void printTwo(int a, int b) {cout << a << b;}

   }

};
...
#include "MyClass_moc.h"  // Code vom MOC (Meta-Object-Compiler) aus "MyClass.h" erzeugt
...
MyClass *x = new MyClass(); MyClass *y = new MyClas();

QObject::connect(x, SIGNAL("twoNumbers(int,int)"), y, SLOT("printTwo(int,int)"));
QObject::connect(x, SIGNAL("anEvent()"), y, SLOT("crashApplication()"));

emit x.twoNumbers(23, 42);
```

Qt

Qt für C++

- Spracherweiterung
  - „`signals`“ – Eventvariablen
  - „`slots`“ – Empfänger (Methoden von Objekten)
- MOC („Meta-Object-Compiler“)
  - Konvertiert signals/slots in normalen C++ Code
  - Gemischt mit einigem an Präprozessortricks (`#define`)
  - Vermeidet „Boilerplate“-Code
  - Rüstet bei der Gelegenheit rudimentäre Reflektion nach
    - Daher der Name „Meta-Object-…“

Qt für Python

- Offiziell unterstützte Platform („Qt for Python“)
  - Früher „PySide“ (es gab/gibt auch „PyQt“)
- Spracherweiterungen nicht nötig
  - Zeiger auf Methoden von Objekten sind Teil der Sprache
  - Delegation daher einfacher als in C++
  - Sicherer, dank GC
  - Besser als C++-Original :-)

Qt 5.11 – Oberklasse QWidget

Properties

Attribute

- acceptDrops : bool
- accessibleDescription : QString
- accessibleName : QString
- autoFillBackground : bool
- baseSize : QSize
- childrenRect : const QRect
- childrenRegion : const QRegion
- contextMenuPolicy : Qt::ContextMenuPolicy
- cursor : QCursor
- enabled : bool
- focus : const bool
- focusPolicy : Qt::FocusPolicy
- font : QFont
- frameGeometry : const QRect
- frameSize : const QSize
- fullScreen : const bool
- geometry : QRect
- height : const int
- inputMethodHints : Qt::InputMethodHints
- isActiveWindow : const bool
- layoutDirection : Qt::LayoutDirection
- locale : QLocale
- maximized : const bool
- maximumHeight : int
- maximumSize : QSize
- maximumWidth : int
- minimized : const bool
- minimumHeight : int
- minimumSize : QSize
- minimumSizeHint : const QSize
- minimumWidth : int
- modal : const bool
- mouseTracking : bool
- normalGeometry : const QRect
- palette : QPalette
- pos : QPoint
- rect : const QRect
- size : QSize
- sizeHint : const QSize
- sizeIncrement : QSize
- sizePolicy : QSizePolicy
- statusTip : QString
- styleSheet : QString
- tabletTracking : bool
- toolTip : QString
- toolTipDuration : int
- updatesEnabled : bool
- visible : bool
- whatsThis : QString
- width : const int
- windowFilePath : QString
- windowFlags : Qt::WindowFlags
- windowIcon : QIcon
- windowModality : Qt::WindowModality
- windowModified : bool
- windowOpacity : double
- windowTitle : QString
- x : const int
- y : const int

Protected Functions

Hierar. Events

virtual void	**actionEvent**(QActionEvent *event)
virtual void	**changeEvent**(QEvent *event)
virtual void	**closeEvent**(QCloseEvent *event)
virtual void	**contextMenuEvent**(QContextMenuEvent *event)
void	**create**(WId window = 0, bool initializeWindow = true, bool destroyOldWindow = true)
void	**destroy**(bool destroyWindow = true, bool destroySubWindows = true)
virtual void	**dragEnterEvent**(QDragEnterEvent *event)
virtual void	**dragLeaveEvent**(QDragLeaveEvent *event)
virtual void	**dragMoveEvent**(QDragMoveEvent *event)
virtual void	**dropEvent**(QDropEvent *event)
virtual void	**enterEvent**(QEvent *event)
virtual void	**focusInEvent**(QFocusEvent *event)
bool	**focusNextChild**()
virtual bool	**focusNextPrevChild**(bool next)
virtual void	**focusOutEvent**(QFocusEvent *event)
bool	**focusPreviousChild**()
virtual void	**hideEvent**(QHideEvent *event)
virtual void	**inputMethodEvent**(QInputMethodEvent *event)
virtual void	**keyPressEvent**(QKeyEvent *event)
virtual void	**keyReleaseEvent**(QKeyEvent *event)
virtual void	**leaveEvent**(QEvent *event)
virtual void	**mouseDoubleClickEvent**(QMouseEvent *event)
virtual void	**mouseMoveEvent**(QMouseEvent *event)
virtual void	**mousePressEvent**(QMouseEvent *event)
virtual void	**mouseReleaseEvent**(QMouseEvent *event)
virtual void	**moveEvent**(QMoveEvent *event)
virtual bool	**nativeEvent**(const QByteArray &eventType, void *message, long *result)
virtual void	**paintEvent**(QPaintEvent *event)
virtual void	**resizeEvent**(QResizeEvent *event)
virtual void	**showEvent**(QShowEvent *event)
virtual void	**tabletEvent**(QTabletEvent *event)
virtual void	**wheelEvent**(QWheelEvent *event)

Qt 5.11 – Oberklasse QWidget

Events mit Signals/Slots (quer zum Baum)

Public Slots

bool	**close**()
void	**hide**()
void	**lower**()
void	**raise**()
void	**repaint**()
void	**setDisabled**(bool *disable*)
void	**setEnabled**(*bool*)
void	**setFocus**()
void	**setHidden**(bool *hidden*)
void	**setStyleSheet**(const QString &*styleSheet*)
virtual void	**setVisible**(bool *visible*)
void	**setWindowModified**(*bool*)
void	**setWindowTitle**(*const QString &*)
void	**show**()
void	**showFullScreen**()
void	**showMaximized**()
void	**showMinimized**()
void	**showNormal**()
void	**update**()

› 1 public slot inherited from QObject

Signals

void	**customContextMenuRequested**(const QPoint &*pos*)
void	**windowIconChanged**(const QIcon &*icon*)
void	**windowTitleChanged**(const QString &*title*)

› 2 signals inherited from QObject

Nachfahre QPushButton (genauer: QAbstractButton)

Signals

void	**clicked**(bool *checked* = false)
void	**pressed**()
void	**released**()
void	**toggled**(bool *checked*)

› 3 signals inherited from QWidget

› 2 signals inherited from QObject

Minimal-Beispiel

Dialog.h

```
#ifndef DIALOG_H
#define DIALOG_H

#include <QDialog>
#include <QPushButton>
#include <QLabel>

class Dialog : public QDialog {
    Q_OBJECT // This macro is needed by MOC
 private:
    QLabel *label;    // contains a text-label
    QPushButton *btn; // contains a button

 public:
    Dialog(QWidget *parent = 0); // Constructor
    ~Dialog();
};

#endif // DIALOG_H
```

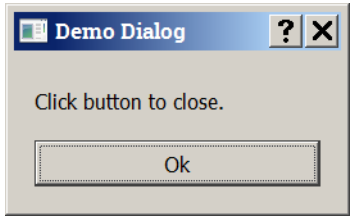
Dialog.cpp

```
#include "dialog.h"
#include <QVBoxLayout>

Dialog::Dialog(QWidget *parent) : QDialog(parent) {
    setWindowTitle("Demo Dialog");
    QVBoxLayout *layout = new QVBoxLayout();
    setLayout(layout);
    label = new QLabel(this);
    label->setText("Click button to close.");
    layout->addWidget(label);
    btn = new QPushButton(this);
    btn->setText("Ok");
    layout->addWidget(btn);
    QObject::connect(
        btn, SIGNAL(pressed()), this, SLOT(close()));
}

Dialog::~Dialog() {}
```

main.cpp

```
#include "dialog.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Dialog w;
    w.show();

    return a.exec();
}
```

Event Loop!



Die Eventhölle

Probleme mit Events

„Widgets“ modellieren zwei Ideen

- Kapselung: Komponenten als „Black Box“
- Reaktion auf Ereignisse

Probleme „Event-Spaghetti-Code“

- Code oft schwer zu verstehen / warten
- Zyklische Eventschleifen (bei starker Kapselung)

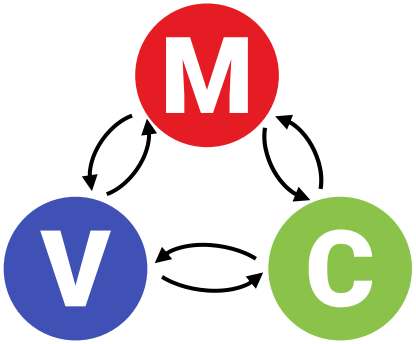
Was tun?

Wir brauchen mehr Struktur!

- MVC – Strukturelle Ordnung
  - Alte Idee, aber bewährt
- FRP – Ordnung des Kontrollflusses
  - Neue Idee, gerade „in vogue“

MVC

Model · View · Controller



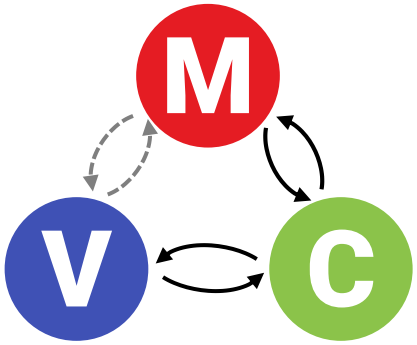
Design
Pattern

GUI Strukture

So far…

- Widgets handle all interaction
  - „Model“: Daten speichern
  - „View“: Daten darstellen
  - „Controller“: Benutzerinteraktion (Maus/Keyboard)
- Begrenzte Wiederverwendung
  - Widgets selbst speichern die Daten
  - Zeichnen/Darstellung muss jedes Mal implementiert werden
  - Benutzerinteraktion muss jedes Mal implementiert werden
- Aufteilung erhöht Modularität

Aufteilung: MVC



„Model“ – Daten speichern

- Modellierung der Daten
- Unabhängig vom UI! (Anwendungsdaten selbst)

„View“ – Daten darstellen

- Bibliothek zum Anzeigen der Daten
- Inklusive z.B.: Buttons, Rahmen, Selektion

„Controller“ – Benutzerinteraktion

- Reagiert auf Ereignisse (Events, Maus/Tastatur)
- Aktualisiert Views, ändert Modell

MVC & „MP“

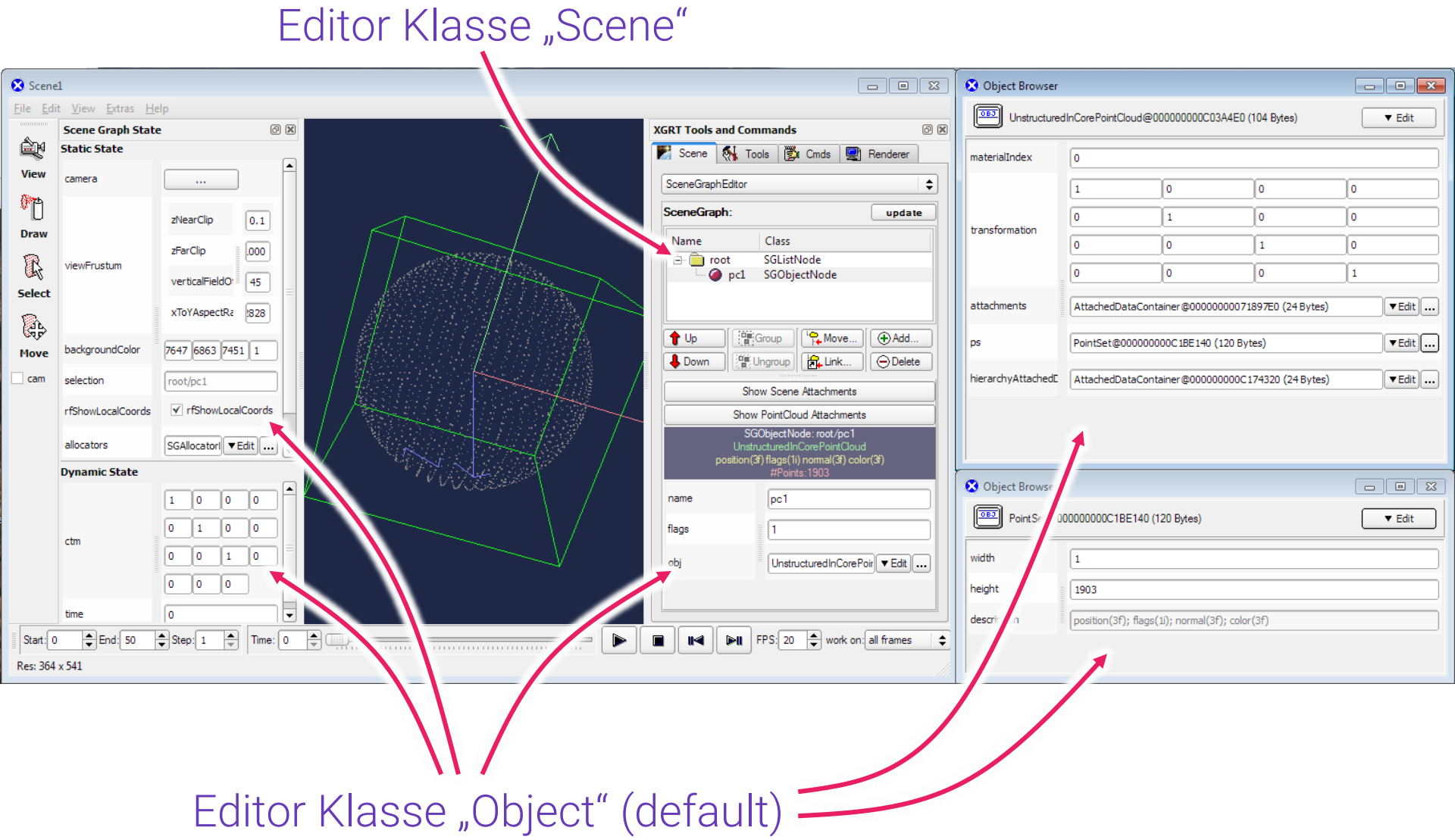
Meta-Pattern

- Viele verschiedene Varianten
- Auch unter leicht unterschiedlichen Namen

Vereinfachte Variante (sehr häufig)

- „Model-Presentation“ oder „Model-Editor“ oder „Model-[View/Controller]“
- Datenmodell separat
- Ein Editorwidget/fenster passend dazu
  - Jede Datenklasse mit einer Editorklasse assoziieren

Beispiel für M-E: GeoX / GeoXL



FRP – Functional Reactive Programming

Allgemein: Reaktive Programmierung

Reactive

- Ist gerade ziemlich hip
- Design Patterns
  - Observer
  - Iterator
  - Visitor

lösen eigentlich alle das gleiche Problem

- Der Kontrollfluss ist anders

Allgemein: Reaktive Programmierung

Reactive

- Ist gerade ziemlich hip
- Design Patterns
  - Observer
  - Iterator
  - Visitor

lösen eigentlich alle das gleiche Problem

- Der Kontrollfluss ist anders

Patterns

Iterator

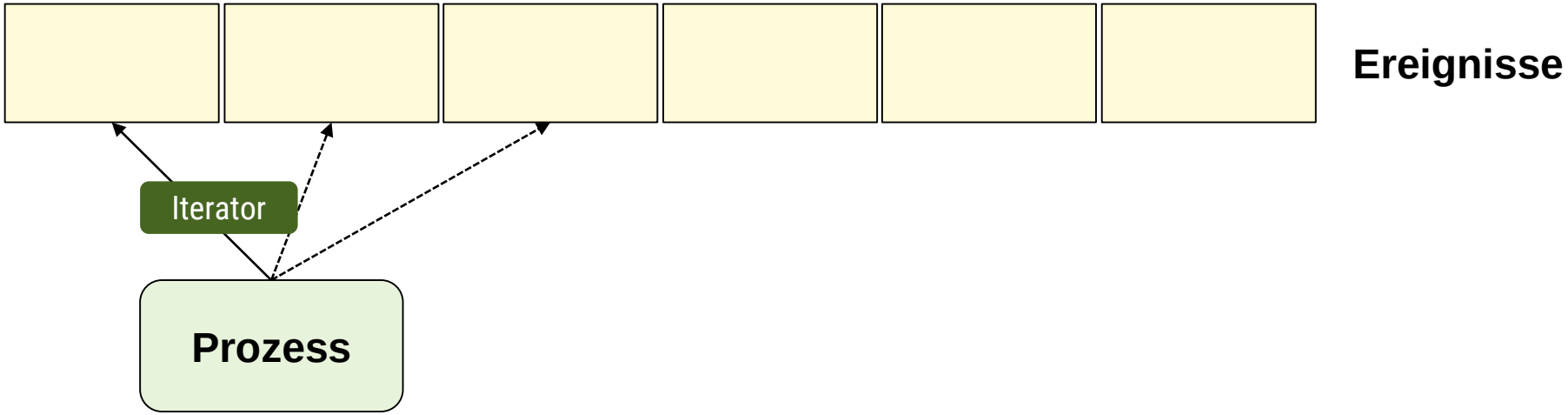
- Laufender Prozess: sucht aktiv durch Elemente
- Kontrolle über Kontrollfluss (frei programmierbar)

Observer

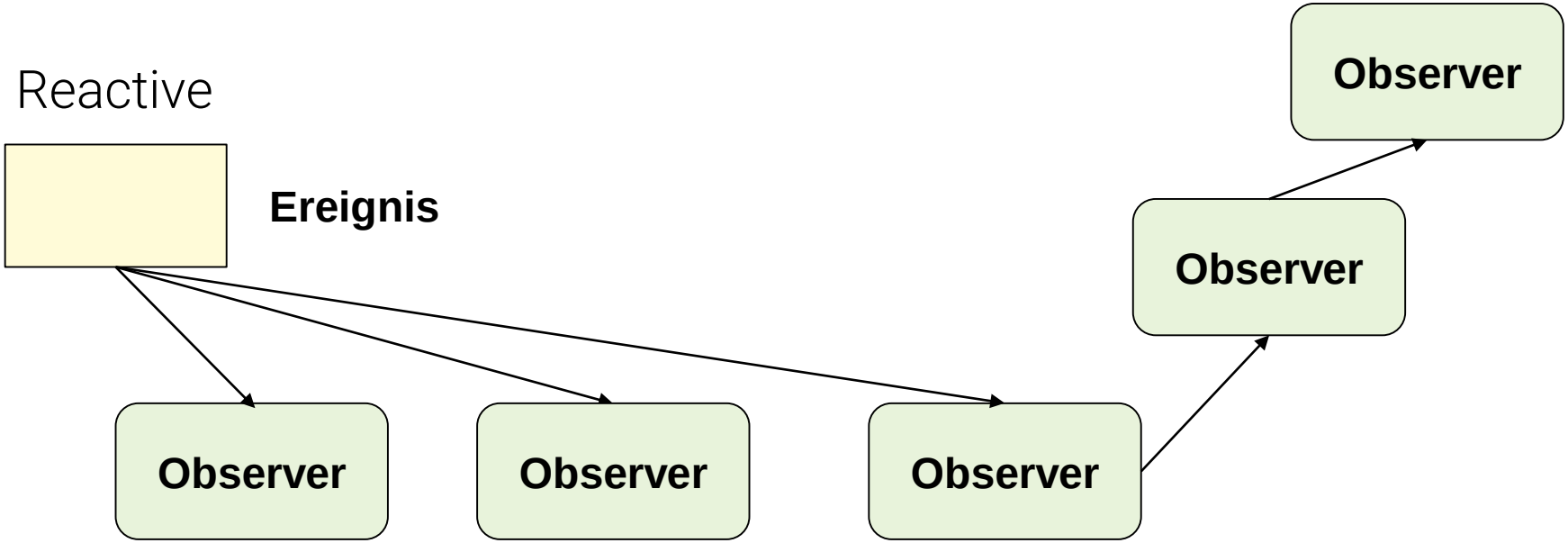
- Laufender Prozess schickt Nachrichten an registrierte Beobachter
- Möglichkeiten
  - Weiterdelegieren
  - Neue Beobachter registrieren / alte entfernen
  - Delegationsketten bauen (→ FPR)

Iterator vs. Observer

Proactive



Reactive



Im GUI-Kontext…

Eventhölle

(Weiteres) Problem mit „Widgets“ und Events

- Spaghetti-Code
- Events können sich zyklisch auslösen
- Oft ungenügende Kapselung
- Strukturiering?

FPR

Moderner Ansatz

- „Functional Reactive Programming“
- Reactive Programming ist eine längere Geschichte
  - Hier nur die Grundidee

Datenflussgraph

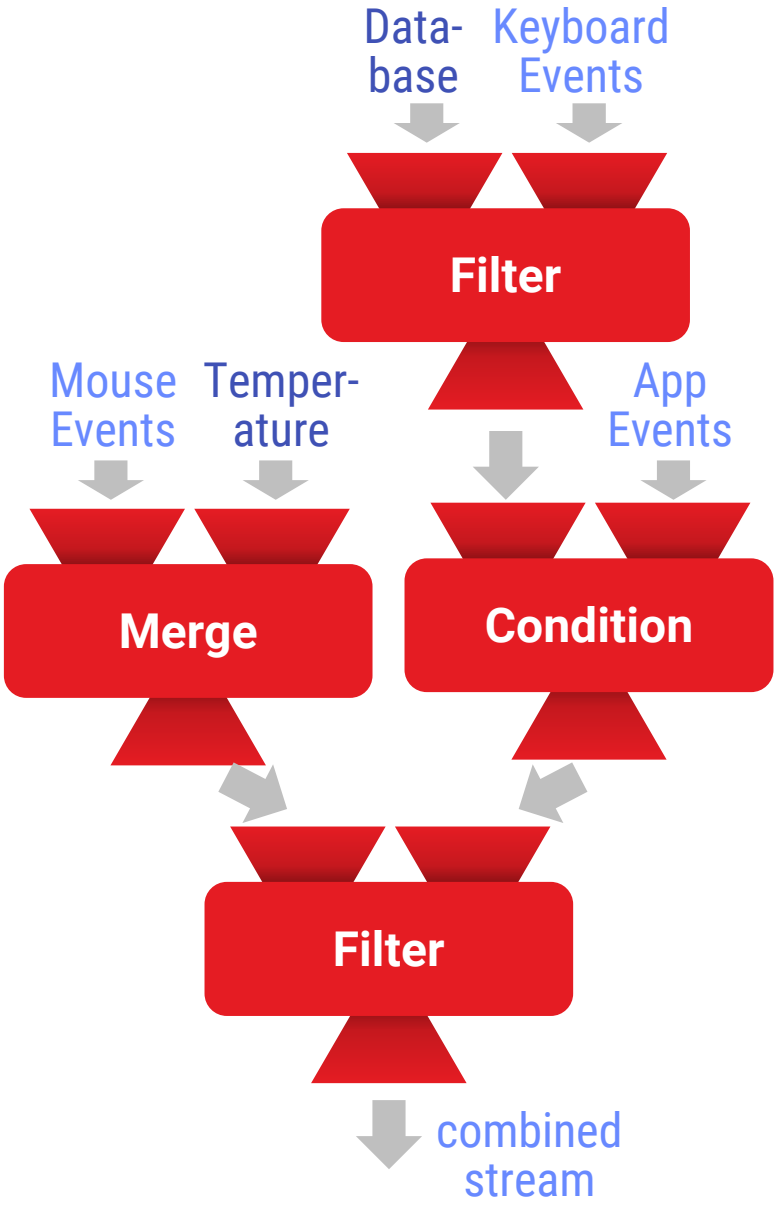
- View ist eine Funktion von Model
- Gleicher Zustand → gleicher View
- Hierarchie von Funktionen die Model in View „übersetzen“

FPR

Was ist mit den Events?

- Controller in Funktional?
- Events als Datenobjekte kapseln
- Ströme von Events in einen Datenflussgraphen einspeisen
- Funktionale Abbildungen: Events (und Folgen davon) werden zu Datentransformationen und View-Updates

Datenflussgraph



Streams

- Events: Things happening
- Values: State of things
- Combined into streams

Event Queue

- One event queue per stream
- Streams of changes to UI

Implementation: Event-Loop!