Johannes Gutenberg-Universität Mainz
Institut für Informatik
Einführung in die Softwareentwicklung
Prof. Dr. Michael Wand
Dr. Christian Ali Mehmeti-Göpel
Visual Computing Group

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Einführung in die Softwareentwicklung

ÜBUNGSBLATT 02: Eine praktische Einführung in die Implementation graphischer Benutzerschnittstellen

05. Mai 2025

Über dieses Übungsblatt

Dieses Übungsblatt umfasst ein kleines Tutorial für die Implementation von GUIs mit Hilfe einer vorgefertigten GUI-Bibliothek. Dies stellt einen Vorgriff auf den Stoff der Vorlesung dar – die Ideen und Architekturmuster hinter solchen Bibliotheken werden wir im Laufe der Vorlesung noch eingehender diskutieren. Um dennoch in den Übungen „echte" interaktive Anwendungen programmieren zu können, wollen wir auf diesem Übungsblatt schon einmal den Einsatz solcher Bibliotheken aus praktischer Perspektive kennenlernen.

Auffallend wird dabei die sogenannte „Ereignisorientierte" Architektur sein: Das Besondere hier ist, dass immer, wenn im GUI etwas passiert (z.B. ein Button wurde gedrückt), ein von der/dem Programmierer/in vorgegebenes Unterprogramm aufgerufen wird, welches dann umgehend wieder beendet werden muss. Der gewohnte durchgängige Ablauf des Programms wird so in kleine Schnipsel zerteilt, was etwas gewöhnungsbedürftig ist (und in der Tat gibt es auch immer noch relativ aktuelle Forschung dazu, wie man das wirklich elegant nutzt, siehe z.B. „reaktive" Ansätze). Uns geht es an dieser Stelle aber erstmal darum, das Programmieren mit ereignisorientierten GUI-Bibliotheken an einem einfachen Beispiel kennenzulernen (wo die Handhabung der Ereignisse nicht schwierig sein sollte). Als Bibliothek werden wir Qt mit Python (via PySide) nutzen. Formal ist folgendes erlaubt:

- Python mit PySide6 für Qt 6 (empfohlen; alle Beispiele und Anleitungen auf diesem Blatt beziehen sich hierauf).
- Scala mit SWING/AWT ist ebenfalls zulässig; wir können hierzu allerdings keine gesonderte Anleitung bereitstellen.

Die Nutzung eines Tools zur Erstellung der Oberflächen (QtDesigner oder ähnliches) ist erlaubt; wir können allerdings auch hier keine Anleitung bereitstellen. Für Neueinsteiger empfehlen wir aber, zumindest einmal auf diesem Übungsblatt alles „per Hand" zu programmieren, um ein Gefühl für die Struktur und das Vorgehen zu bekommen.

Empfohlene Quellen / Hilfsmittel:

Dokumentation zu Qt for Python (PySide 6):

https://doc.qt.io/qtforpython-6/

Inhalt des Aufgabenblattes

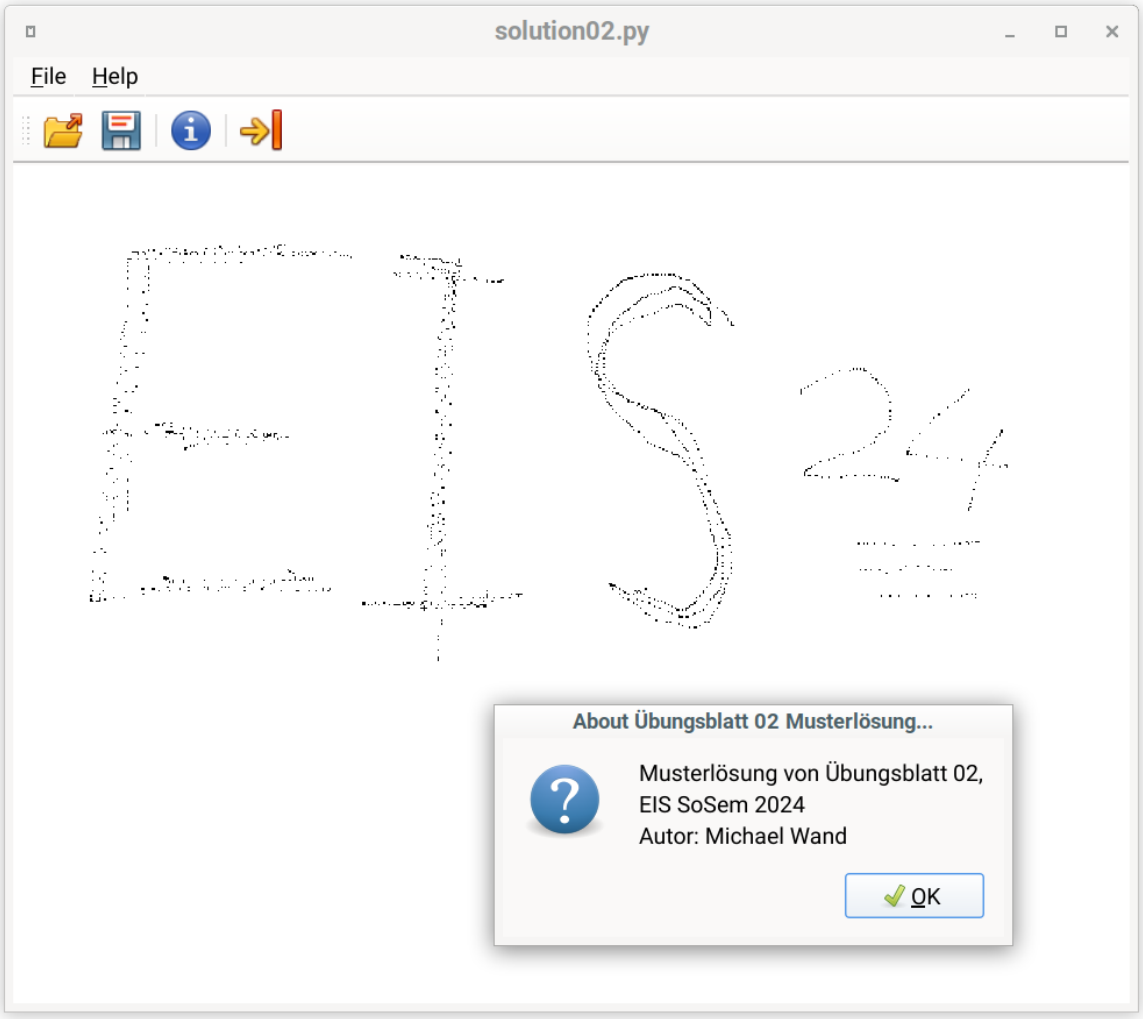
Wir programmieren ein einfaches Pixel-basiertes Malprogramm, bei dem man mit der Maus Pixel einfärben kann. An dem Beispiel kann man alle Schritte und Komponenten einer „richtigen" Anwendung kennenlernen, aber die Sache ist trotzdem verhältnismäßig einfach zu realisieren.

Abgabe: (Abgabe 5. SW)

Für dieses Übungsblatt ist nur eine Woche Bearbeitungszeit vorgesehen. Das Übungsblatt muss entsprechend bis zum **11. Mai 2025, 23:59h** in LMS abgegeben werden. Laden Sie dazu den Source-Code aller Aufgaben hoch. Die Ergebnisse müssen in den Übungen zwischen dem 12.-16. Mai 2025 vorgestellt werden.

Aufgabe 1: Installieren Sie die Tools für die folgenden Programmiersprachen auf Ihrem Rechner

Bewertung: 40+10+30+20 = 100 Punkte



(Screenshot von der Musterlösung)

Nutzen Sie im Folgenden die GUI-Bibliotheken Qt in Verbindung mit der Programmiersprache Python oder SWING/AWT in Verbindung mit Scala um die folgenden Aufgaben zu lösen. Wichtig: Am Ende stehen recht umfangreiche Erläuterungen für die Lösung mit Qt6+Python (empfohlener Lösungsweg); die Lösung sollte damit wesentlich erleichtert werden.

(a) Erzeugen Sie ein Hauptfenster, welches mit einem Dropdown-Menü (Hauptmenue) ausgestattet ist, welches die folgenden Menüpunkte und Untermenüpunkte bietet:

- Datei
    - Öffnen..
    - Speichern Als…
    - Beenden…
- Hilfe
    - Informationen…

„Beenden…": Die Auswahl des „Beenden"-Menüpunktes soll die Anwendung beenden. Bevor die Anwendung geschlossen wird, soll aber noch ein Dialogfenster erscheinen, mit einer Sicherheitsabfrage, ob man wirklich die Anwendung schließen möchte (mit Auswahl z.B. JA/NEIN, oder Schließen/Abbrechen). Nur wenn die/der Anwender/in die Auswahl bestätigt, soll das Programm wirklich beendet werden.

„Informationen…": Die Auswahl des Menüpunktes „Informationen" soll einen Dialog anzeigen, der einen Einfachen Informationstext ausgibt. Vorschlag: „Übungsblatt 02, EIS SoSem 2025, <Name der Autorinnen/Autoren>". Der Dialog soll sich mit einem Knopf wieder schließen lassen.

Optional (keine Punkte): können Sie die Menüs „schöner" machen, indem Sie Icons, „Accellerators" (unterstrichene Buchstaben als Short-Cuts mit „alt") und Keybord-Shortcuts hinzufügen (z.B. Ctrl+O für Öffnen oder Ctrl+Q für beenden).

Wichtig: Die genaue Benennung und die Sprache (deutsch oder englisch) spielt für die Bewertung keine Rolle, solange klar verständlich ist, was gemeint ist (die Musterlösung ist z.B. auf Englisch).

(b) Fügen Sie dem Hauptfenster eine „Toolbar" hinzu, also eine Leiste mit Knöpfen, mit denen die gleichen Aktionen ausgelöst werden können. Hier sollte zumindest ein Text auf den Knöpfen erscheinen; Sie können aber auch Icons benutzen (siehe Screenshot Musterlösung).

(c) Fügen Sie nun eine Zeichenfläche hinzu, auf der man malen kann, indem man mit der Maus (bei gedrückter Maustaster) herüberfährt (siehe Musterlösung). Das Bild soll erhalten bleiben, auch wenn das Fenster zwischenzeitlich verdeckt oder minimiert wurde. Hierzu ist es nötig, das Bild, auf dem Sie malen selbst zu speichern, z.B. mit Hilfe eines `QImage` (siehe Erläuterungen).

(d) Füllen Sie zum Schluss die noch fehlenden Funktionen ein: Mit „Speichern Als…" wird das gemalte Bild in eine Datei im PNG-Format gespeichert, und mit „Öffnen" kann man ein bereits erstelltes Bild wieder einladen (`QImage` bietet direkt diese Funktionalität).

Hinweise zur Lösung

Hinweise zu (a)

Erstellen eines Hauptprogramms. In der Musterlösung waren die folgenden Import-Anweisungen nützlich (kann man copy + pasten):

```
from PySide6.QtCore import Qt
from PySide6.QtGui import QKeySequence, QPixmap, QCloseEvent, QImage
from PySide6.QtGui import QPainter, QColor
from PySide6.QtWidgets import QApplication, QMenuBar, QToolBar
from PySide6.QtWidgets import QWidget, QFileDialog
from PySide6.QtWidgets import QMainWindow, QMessageBox
```

Als Rahmen für das Programm kann man mit dem Code aus Übungsblatt 01 starten, und eine eigene Klasse für das Hauptfenster definieren (ableiten von `QMainWindow`). Für Aufgabenteil c) braucht man auch noch eine eigene Widget-Klasse (abgeleitet von `QWidget`):

```
# Our own widget type (for later)
class MyPaintArea(QWidget):
    def __init__(self, parent: QWidget):
        super().__init__(parent)


# Our own Main Window type
class MyWindow(QMainWindow):
    def __init__(self, parent: QWidget):
        # call parent constructor (required for QMainWindow)
        super().__init__(parent)


# our main program starts here, Python-style
if __name__ == "__main__":
    # create an application object (needs cmd-line arguments)
    app: QApplication = QApplication(sys.argv)

    # Create the main window.
    main_window: MyWindow = MyWindow(None)
    main_window.show()

    # Start the event loop.
    # Ends only after closing the main window
    app.exec()
```

Man kann nun GUI-Elemente für die eigene „`MyWidow`"-Klasse anlegen. Am besten macht man das direkt im Konstruktor („`__init__`"). Grundsätzlich empfehle ich (so habe ich es in fast allen Programmen und Frameworks gesehen) immer ein neues Feld in der `MyWindow`-Instanz anzulegen, indem eine Referenz auf das GUI-Element hinterlegt wird (um später nochmal einfach drauf zugreifen zu können). Zusätzlich muss man das GUI-Element in das Qt-GUI „einhängen". Im Beispielcode von Übungsblatt 01 ist das z.B. für die Buttons wie folgt geschehen:

```
# Hier wird einfach nur die Referenz angelegt
# (Qt ignoriert das; das ist nur für uns)
self.my_frame = QFrame(self) # Das Argument bezeichnet den „Besitzer“ (parent)
                             # Hier ist das unser Hauptfenster selbst
# Nun können wir das Widget (einen einfachen Leeren Container mit Rahmen
# für weitere Widgets) in das Hauptfenster „einhängen“;
# in diesem Fall geht das so:
self.setCentralWidget(self.my_frame)
```

Ähnlich können wir die Element anlegen, die wir für unser GUI brauchen dazu ein paar Tipps/Code-Schnipsel aus der Musterlösung (Details: Siehe Qt6 for Python Dokumentation). Auch diese Befehle sollte in `MyWindow.__init__(self, parent)` stehen:

```
# Die drei Punkte am Ende benutzt man, falls noch weitere Dialoge folgen
# (das ist nur eine ästhetische Konvention; Qt ist das egal).
self.cool_menu: QMenuBar = self.menuBar().addMenu("Cool...")
```

Nun kann man Untermenüs als „Aktionen“ (`QAction`) hinzufügen[1]:

```
self.fancy_action = self.cool_menu.addAction("Do Something Fancy...")
self.fancy_action.setShortcut(QKeySequence(Qt.CTRL | Qt.Key_O))
self.fancy_action.setIcon(QPixmap("dateiname_für_icon.png"))
self.fancy_action.triggered.connect(self.perform_fancy_action)
```

Shortcuts und Icons sind natürlich optional. Die letzte Zeile verbindet das Auslösen der Aktion (Menüpunkt ausgewählt) mit dem Aufruf der Methode „`perform_fancy_action`“ im eigenen Objekt. Diese muss man natürlich entsprechend definieren, also etwa so:

```
# Our own Main Window type
class MyWindow(QMainWindow):
    def __init__(self, parent: QWidget):
        # ...

    def perform_fancy_action(self):
        # ... dieser Code wird vom Menü aufgerufen...
```

[1] Noch ein optionaler Tipp/Hinweis: Bei den Menütexten bei `addMenu`/`addAction` kann man „Accelerators“ (Unterstriche unter Buchstaben um diese als Shortcut oder mit der „Alt“-Taste zu benutzen“ hinzufügen, indem man ein „&“-Zeichen dem Buchstaben vorstellt. Also z.B. `addAction("Save &As…")` für Save As…
Nicht wichtig, aber hübscher und es erleichtert in einer echten Applikation den Bedienung. :-)

Hinweise zu (b)

Man kann nun auch ganz einfach eine „Toolbar“ hinzufügen, auf der die gleichen Aktionen verfügbar gemacht werden. Dies wird wieder im Konstruktor `MyWindow.__init__()` erledigt:

```
self.nice_toolbar: QToolBar = self.addToolBar("Some Nice Tools")
self.nice_toolbar.addAction(self.fancy_action)
```

An dieser Stelle nutzen wir die Tatsache, dass wir uns einen Verweis auf „`fancy_action`“ vorher schon im eigenen Objekt gemerkt haben.

Hinweise zu (c)

Um eine Malfläche zu bauen, ist etwas Aufwand nötig (insbesondere, wenn man es vernünftig machen möchte – hier schauen wir uns eine halbwegs „ordentliche“ Lösung an). Dies ist sicher der schwierigste Teil der Aufgabe.

Die Idee ist es, ein eigenes „Widget“ zu programmieren, welches einen eigenen Zustand hat (es speichert das Bild, das gerade gemalt wird) und welches auf Ereignisse so reagiert, wie wir es möchten.

Ein eigenes Widget zu definieren ist einfach: Wir fügen an geeigneter Stelle (s.o.) eine Definition für eine Klasse an, die das bestehende QWidget durch ableiten verfeinert (was da genau dahintersteckt, werden wir in der Vorlesung natürlich noch ausführlich besprechen). Im Konstruktor („`__init__`“) wird ein Feld angelegt, indem wir unser Bild speichern. Ich habe es hier „`image`“ genannt.

```
class MyPaintArea(QWidget):
    def __init__(self, parent: QWidget):
        super().__init__(parent)
        self.setMinimumWidth(640) # Größe des neuen Widgets muss man
        self.setMinimumHeight(480) # unbedingt selbst einstellen!
        self.image: QImage = QImage(640, 480, QImage.Format_RGB32) # 640x480pix
        self.image.fill(QColor(255, 255, 255)) # mit weiß löschen
        # ... man braucht sicher noch mehr ;-) ...

    # Diese Methode wird immer aufgerufen, wenn das GUI-Element auf dem Bild-
    # schirm dargestellt warden soll. Man programmiert hier, wie es sich
    # "selbst zeichnen" soll. Einfach mal ausprobieren!
    def paintEvent(self, event):
        painter: QPainter = QPainter(self)
        # hier kann man nun auf das Widget zeichnen
        # Sinnvoll ist z.B.
        # painter.drawImage(<x-Koord: int>., <y-Koord: int>., <ein QImage>)
        # oder zum Testen painter.fillRect(…), drawLine(…) o.ä.
        # (siehe Doku der Klasse QPainter in PySide6)

    # Diese Methode wird jedes Mal aufgerufen, wenn die Maus über unserem
    # GUI-Element gedrückt wird
    def mousePressEvent(self, event):
        # Mit einer solchen Abfrage kann man überprüfen, welcher Knopf es war
        if event.button() == Qt.MouseButtons.LeftButton:
            # ...
```

```
    # Hier das gleiche für das Loslassen von Maustasten
    def mouseReleaseEvent(self, event):
        # ...

    # Und diese Methode wird aufgerufen, falls die Maus über unserem
    # GUI-Element bewegt wird. Achtung: Man muss natürlich prüfen, ob die
    # Taste gedrückt wurde...
    def mouseMoveEvent(self, event):
        # ...
```

Hinweise zu (d)

Nützliche Funktionen für Dateioperationen:

Die folgenden beiden Funktionen fragen den Benutzer nach einem Dateinamen für Laden bzw. Speichern:

```
file_name, selected_filter = QFileDialog.getOpenFileName(parent, "Title", "",
"PNG Files (*.png)")

file_name, selected_filter = QFileDialog.getSaveFileName(parent, "Title", "",
"PNG Files (*.png)")
```

Die Parameter sind wie folgt:

`parent` – Referenz auf Besitzer / übergeordnetes Fenster (typ. das eigene MainWindow; in `__init__` nutzt man hier `self`)

`"Title"` – Ansprache / Title der Dialogbox

`""` – der dritte Parameter ist das Verzeichnis, in dem wir als erstes nachschauen

`"PNG Files (*.png)"` – „Filter" mit Angabe des Dateityps

Rückgabe sind zwei Werte (Tupel von zwei Strings):

- `file_name` (der Dateiname, leer, falls abgebrochen wurde)
- `selected_filter` (gewählter Dateityp, falls einen das interessiert)

Es ist auch noch gut zu wissen, dass der Konstruktor von QImage wie folgt benutzt werden kann:

`QImage(file_name)` lädt direkt die angegebene Bilddatei ein (z.B. `file_name = "test.png"`).

Auch das Speichern ist kein Problem:

```
img: QImage = QImage(640, 480, QImage.Format_RGB32)
# ...
img.save("my_image.png")
```

Der letzte Befehl legt das Bild in einer Datei ab (Dateityp wird nach der angegebenen Endung automatisch gewählt).