

GUI-Programmierung in Java

Rahel Lanz, Patrick Bucher
12. Dezember 2016

Ablauf

1) OOP-Folien und Beispiele

- a) Rahel Lanz: GUI-Komponenten, Layout-Manager, Ereignisgesteuerte Programmierung
- b) Patrick Bucher: Plattformunabhängigkeit, Toolkits allgemein und Swing im Speziellen, Code-Beispiel in Swing
- c) Rahel Lanz: JavaFX, MVC, Code-Beispiel in JavaFX
- d) Patrick Bucher: GUI-Designer, Demo NetBeans GUI-Builder (Swing)
- e) Rahel Lanz: Demo JavaFX GUI-Designer
- f) Patrick Bucher: Empfehlungen und Zusammenfassung

2) Eigene Beispiele

- a) Rahel Lanz: Refactoring einer JavaFX-Applikation bestehend aus einer einzigen Klasse nach MVC
- b) Patrick Bucher: Demonstration des State-Patterns anhand eines Texteditors (mit Aufgaben)

GUI – Graphical User Interface

GUI – Grundlagen: Beispiel "Rechner"

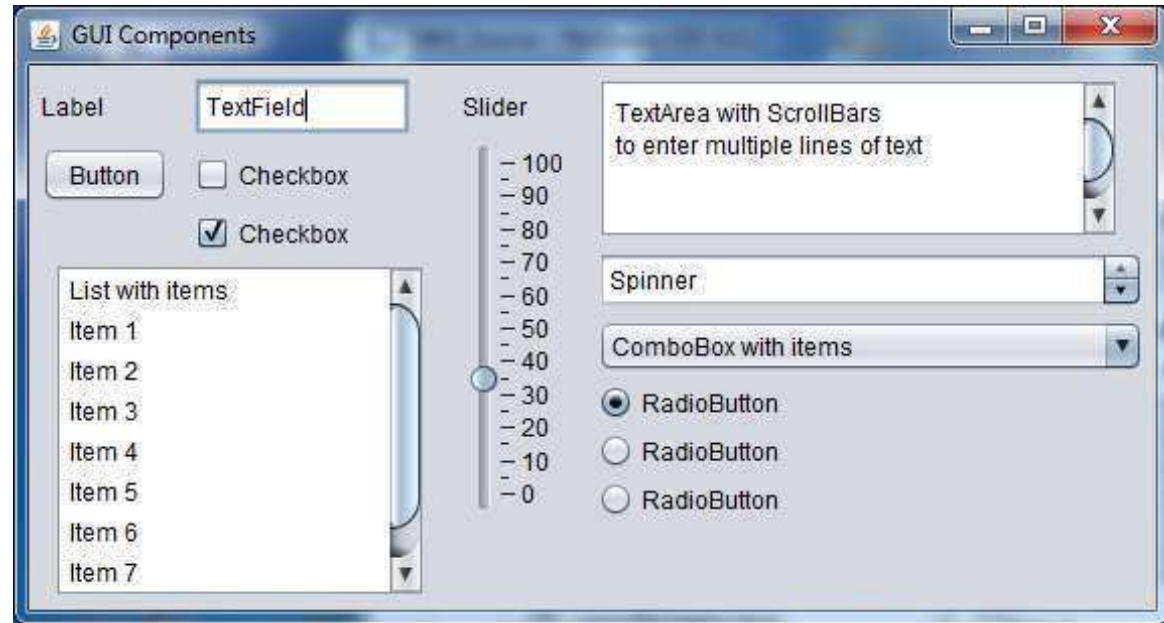
- GUIs bestehen in der Regel aus einer Vielzahl von einzelnen Elementen → sogenannte GUI-Komponenten (Objekte).
- GUI-Komponenten passen hervorragend in die objektorientierte Programmierung: Sie haben einen Zustand und ein Verhalten.
 - Lassen sich sehr gut wiederverwenden.
- Beispiel eines einfachen Rechners:
 - **1** Window / Frame
 - **1** Menu mit **n** MenuItems (hierarchisch)
 - **1** Text Field für Ausgabe des Resultates
 - **28** verschiedene Buttons
 - unterschiedliche Position, Grösse, Beschriftung und ausgelöste Aktion, aber immer **ein** Button



Elementare GUI-Komponenten - Beispiele

- Ausgewählte Basiskomponenten (im Beispiel sichtbar)

- Frame / Window
- Label
- TextField
- Button
- List
- Checkbox
- Radiobutton
- Slider



- Weitere bekannte Elemente:

- Menu, Table, TreeView, Tab etc. ➔ laufend neue Elemente!

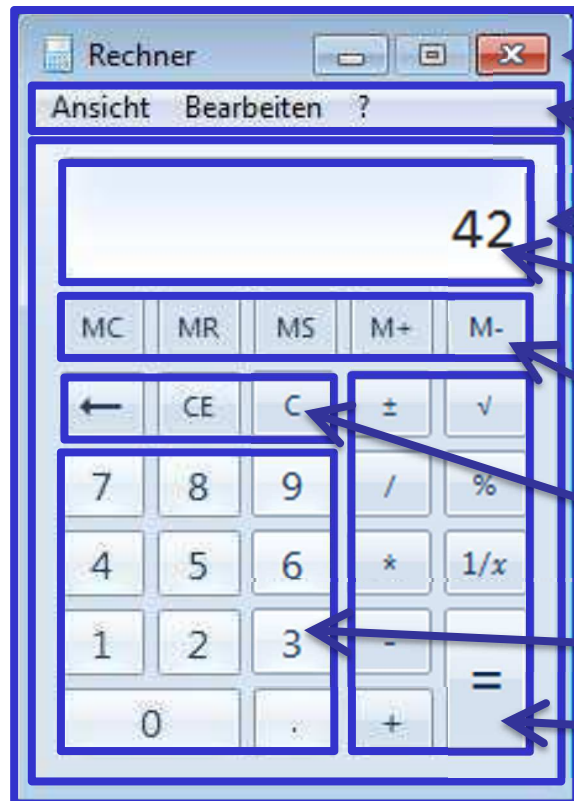
- Nicht alle GUI-Elemente sind in allen Betriebssystemen vorhanden und verhalten sich auch nicht zwingend genau gleich!

Hierarchische Strukturierung von GUI's

- Bei GUIs ist sehr typisch, dass was eigentlich (relativ) «flach» aussieht, im Hintergrund stark hierarchisch strukturiert ist.
- Dafür unterscheidet man zwischen wirklich sichtbaren Teilen (die eigentlichen GUI-Komponenten) und sogenannten **Containern**.
- Container (Behälter) können eine beliebige Anzahl Komponenten (die auch wieder Container sein können!) enthalten.
 - ➔ Composite Pattern, GoF, mehr dazu im Modul VSK
- Beispiele für Container sind
 - Window (Frame) – die Grundfläche einer GUI-Applikation.
 - Panels – eine (nicht zwingend visuelle) Hilfsfläche zur Zusammenfassung von Einzelkomponenten.

Beispiel: Rechner-GUI – Konzeptionelle Idee

- Das GUI eines einfachen Taschenrechners könnte z.B. wie folgt strukturiert sein:



- Fenster (Frame)

- Menu-Bar

- Content-Panel (Root Panel)

- Display-Panel (mit TextField)

- Eingabebereich (je: Panel mit Buttons)

- Memory-Control

- Input-Control

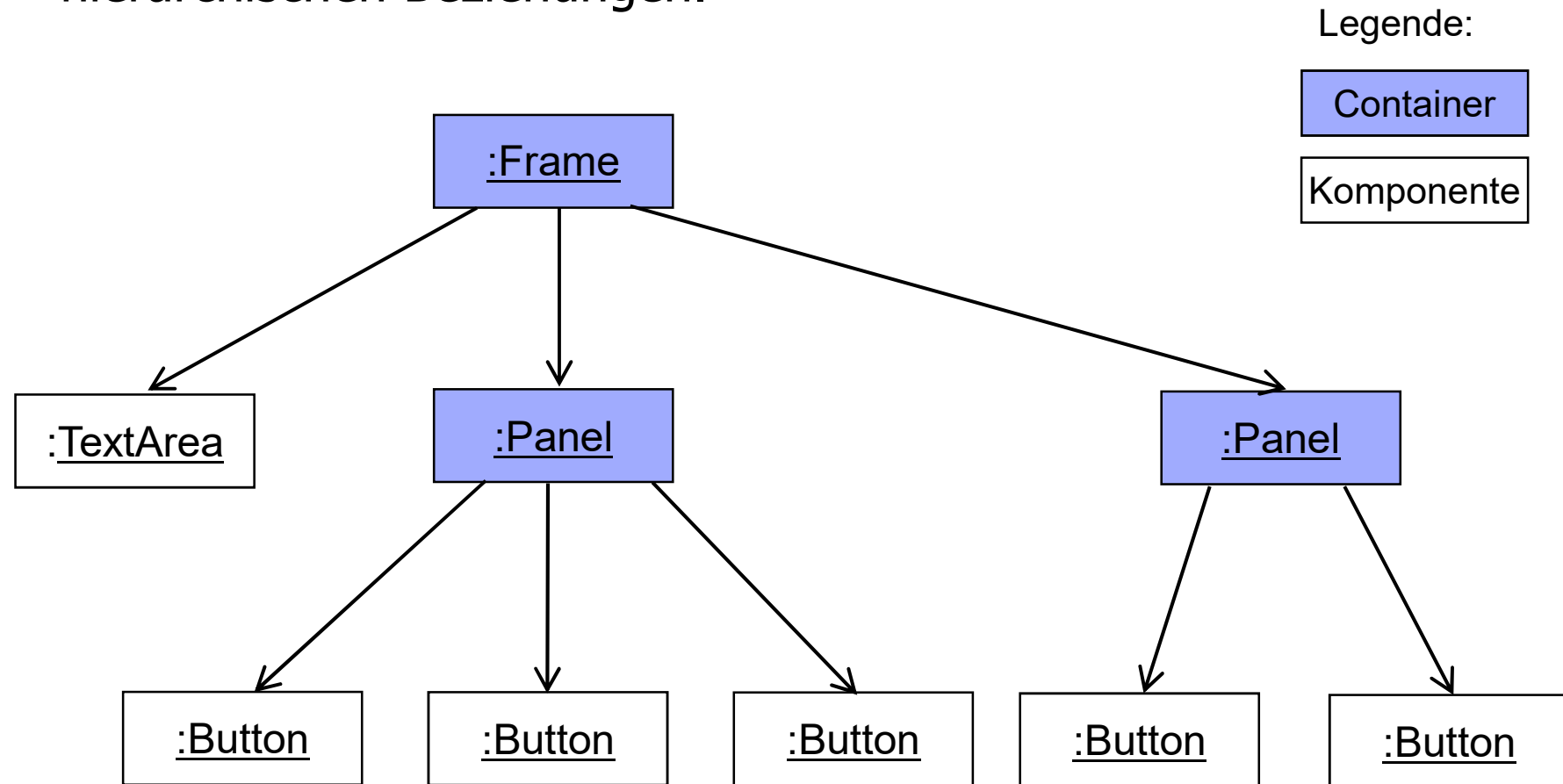
- Dezimal-Pad

- Operation-Control

➔ Hierarchische Struktur

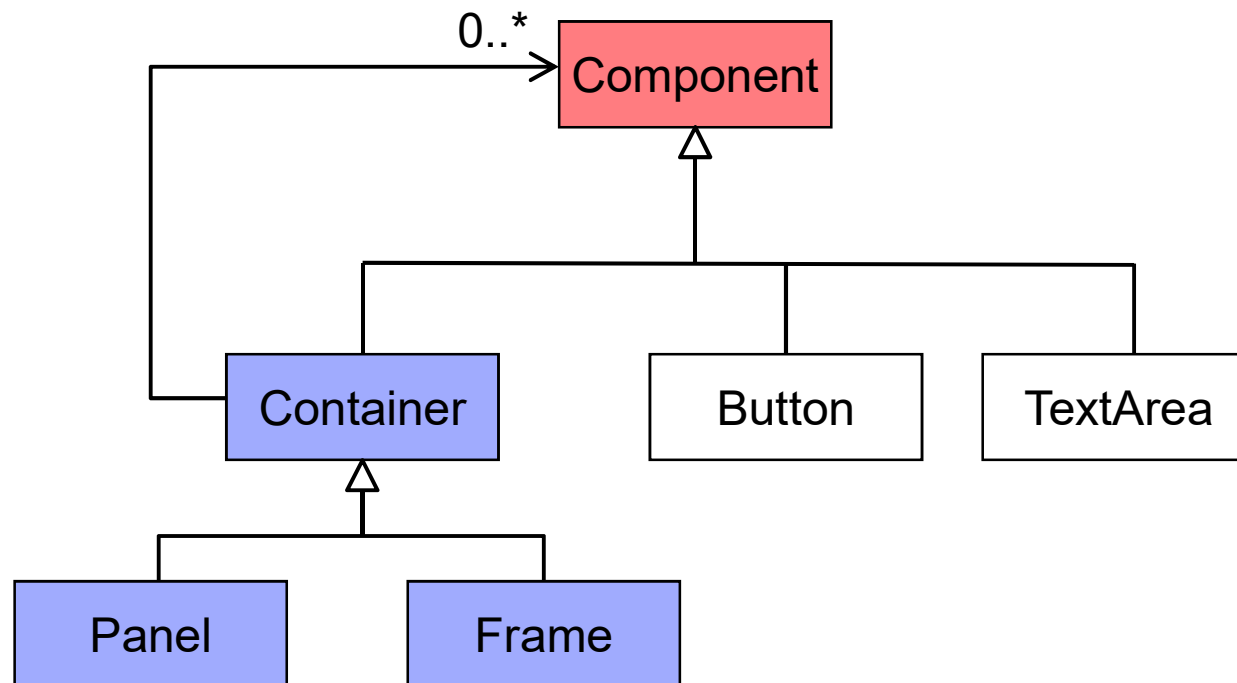
Objektdiagramm (Konzeptionell)

- Reduziertes Diagramm, zur Illustration der hierarchischen Beziehungen.



Klassendiagramm (Konzeptionell)

- Composite-Pattern (Kompositum) nach GoF:
Ein Container kann Komponenten oder aber wiederum Container enthalten.



Legende:

Basisklasse

Container

Komponente

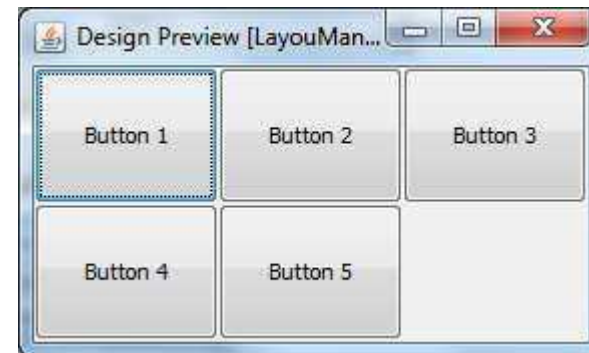
Layout-Konzept

Positionierung von Komponenten in einem Dialog

- Fenstersysteme haben sehr unterschiedliche Desktop-Auflösungen, Schriftgrößen und -arten, Abstände, Ränder usw.
- Pixelgenaue Positionierung und Dimensionierung der Komponenten ist längst **keine adäquate Lösung** mehr!
 - In den 90er-Jahren war das z.B. auf Windows unter Standard-VGA, SVGA oder XGA-Auflösungen durchaus üblich (96dpi).
 - Ebenso erinnern sich manche vielleicht noch an die 1-Pixel-GIFs im Webbereich (Pre-CSS-Ära).
- **Plattformübergreifend** (verschiedene OS) und mit HiDPI-Bildschirmen (z.B. Retina Display von Apple) wird es aber sehr schwierig: Skalierungen von bis zu 200% und mehr!
- Lösung: Konzept der Layouts mit Constraints.

Konzept der Layout-Manager

- Man «beschreibt» die Anordnung der einzelnen Elemente durch die Organisation mittels sogenannter Layouts. Drei **Beispiele**:
- **Flow**-Layout - ein Element nach/neben dem Anderen.
- Horizontal oder Vertikal
- **Grid**-Layout: Ein- und Anordnung der Elemente in eine virtuelle Gitterstruktur.
- **Border**-Layout: Aufteilung in verschiedene Regionen, die bei einem Resize nicht alle gleichmässig mitwachsen.



Layout-Manager

- Die verschiedenen Layout-Manager werden häufig mit- und ineinander (hierarchisch) verknüpft.
- Können mit Bedingungen (Constraints) konfiguriert werden, welche Abstände und Ränder für und um Komponenten festlegen.
- Jedes Element hat eine minimale, optimale und maximale Grösse, abhängig von verwendeten Schriften und Auflösungen.
- Die Elemente organisieren sich innerhalb des Layouts selber und adaptieren ihre Grösse bei einem Resize (Grössenveränderung z.B. eines Dialoges/Fensters)
- Herausforderung: Benötigt einiges an Planung und Erfahrung!
 - Kein schnelles «pinseln» von Dialogen!


Ereignisgesteuerte Programmierung

Ereignisgesteuerte Programmierung

- Sehr typische Technik für GUI-Programmierung (aber nicht nur).
- Die meiste Zeit wartet das Programm.
- Erst wenn ein Ereignis (Event) eintritt, geschieht etwas.
 - Tastendruck, Mausklick, Touchgeste, Fusstritt etc.
- Das Ereignis löst dann eine (vor-) bestimmte Aktion aus.
 - Reihenfolge der Ereignisse kann völlig frei wählbar sein, oder aber auch Schrittweise zu einem Ziel führen.
- Effektive Bearbeitung kann auch in Hintergrund(-prozessen) stattfinden.



Ereignisgesteuerte Programmierung: Event/Listener!

- Die GUI-Programmierung funktioniert in Java massgeblich mit dem bereits bekannten Observer-Pattern per Event/Listener-Modell.
 - Dadurch erreicht man eine möglichst lose Kopplung zwischen dem Auslöser einer Aktion und der Aktion selber.
- Ein populärer (GUI-)Event in Java ist z.B. der **ActionListener**, der für unmittelbare Aktionen wie z.B. das Drücken eine Buttons genutzt wird:
 - Event-Quelle → z.B. **Button**-Objekt 
 - Event «click» → **ActionEvent**-Objekt wird gefeuert.
 - Event-Verarbeitung → Beliebiges Objekt welches das **ActionListener**-Interface implementiert hat, und in der Methode **actionPerformed(...)** auf das Ereignis reagieren kann.

GUI-Programmierung: Spezielle Herausforderung bei Java

Plattformunabhängigkeit von Java

- Java ist Plattformunabhängig, und läuft dank der Java Virtual Machine (JVM) auf vielen, verschiedenen OS-Plattformen.
 - Abstraktion des ganzen Betriebssystems mit Prozessen, Speicherverwaltung, Dateisystemen, I/O und auch des **GUI**!
- Im Bereich der grafischen Schnittstellen gibt/gab es zwischen den unterschiedlichen Betriebssystemen aber beinahe unüberwindbare Hürden!
- Es existieren unzählige, verschiedenste Window-Manager/Desktops:
 - Apple: Platinum, Aqua (verschiedene Generationen)
 - Unix/Linux: Motif, KDE, Gnome, Mate, Compiz,... (Dutzende!)
 - Windows: WPF (verschiedene Generationen)
- Wie bringt man das alles unter einen Hut?

Verschiedene Plattformen



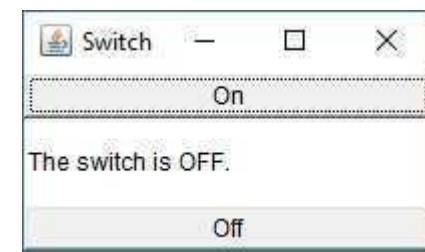
GUI – Frameworks in Java

Java kennt mittlerweile **drei** verschiedene (und teilweise aufeinander aufbauende) GUI-Frameworks, die auch je in verschiedenen Generationen existieren.

- AWT
 - Seit Java 1.0
- Swing
 - Seit Java 2 (1.2)
 - Weiterentwicklung inzwischen eingestellt (nur Wartung).
 - Derzeit grösste Verbreitung.
- JavaFX
 - Seit Java 8 (1.8) im JDK integriert
 - Wird seit 2008 entwickelt → neustes, modernstes Framework.

Abstract Window Toolkit (AWT)

- AWT «wrapped» die nativen GUI-Komponenten des jeweiligen Betriebssystems!
 - Dadurch relativ schlank, einfach und effizient.
 - Look&Feel entspricht voll der jeweiligen Zielplattform.
- Hauptproblem: Weil Java-Programme auf **allen** Plattformen laufen müssen, wird nur die **kleinste gemeinsame Menge** der auf allen Plattformen verfügbaren GUI-Komponenten unterstützt!
 - Somit nur sehr eingeschränkte Möglichkeiten.
 - Auf einzelnen Plattformen fehlen evtl. für diese sehr typische und essentielle Komponenten.

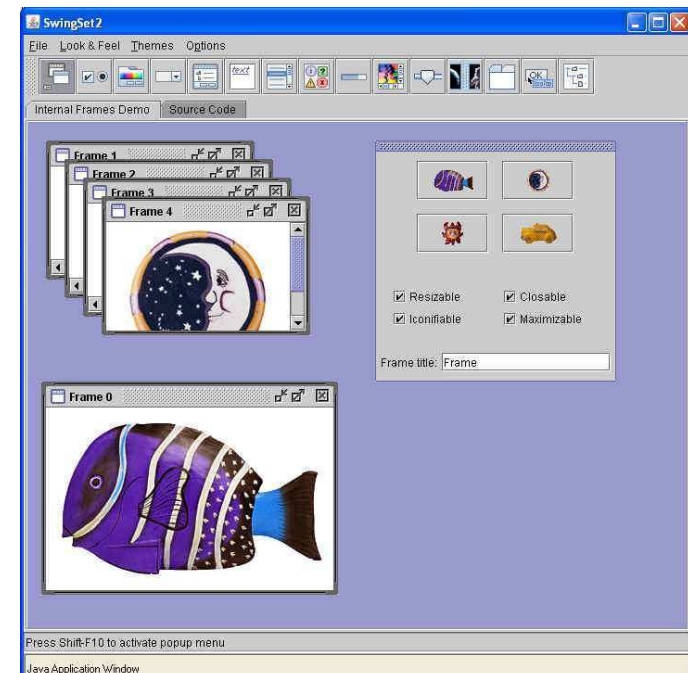


AWT – Die Basistechnologie im Hintergrund

- AWT stellt als solches zwar noch immer eine wichtige Basistechnologie für die GUI-Programmierung mit Java dar, hat aber als eigentliche Plattform für neue GUI's keine Bedeutung mehr.
- Konkret: Programmieren Sie keine neuen GUI's nur auf Basis von AWT, sondern verwenden Sie die darauf aufbauenden GUI-Frameworks wie **Swing** (alt) oder **JavaFX** (neu).

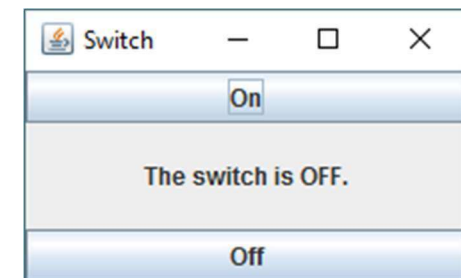
Swing – die Anfänge

- Mit Java 2 (Version 1.2) wurde ein komplett neues GUI-Konzept realisiert: Java implementiert ein **eigenes Set** von umfangreichen leichtgewichtigen GUI-Komponenten.
 - Quasi eine eigene Oberfläche, mit eigenem Look&Feel (Metal).
 - Auf **allen** Plattformen sind somit alle Komponenten verfügbar.
 - Basiert selber auf wenigen, fundamentalen AWT-Klassen.
- Java-Applikationen sahen nun auf allen Plattformen identisch aus (ausser die Fensterdekoration) und waren klar als Java-Applikation erkennbar.
 - Metal-Style war sehr umstritten.
 - BenutzerInnen verlangten bald nach OS-Look&Feel.

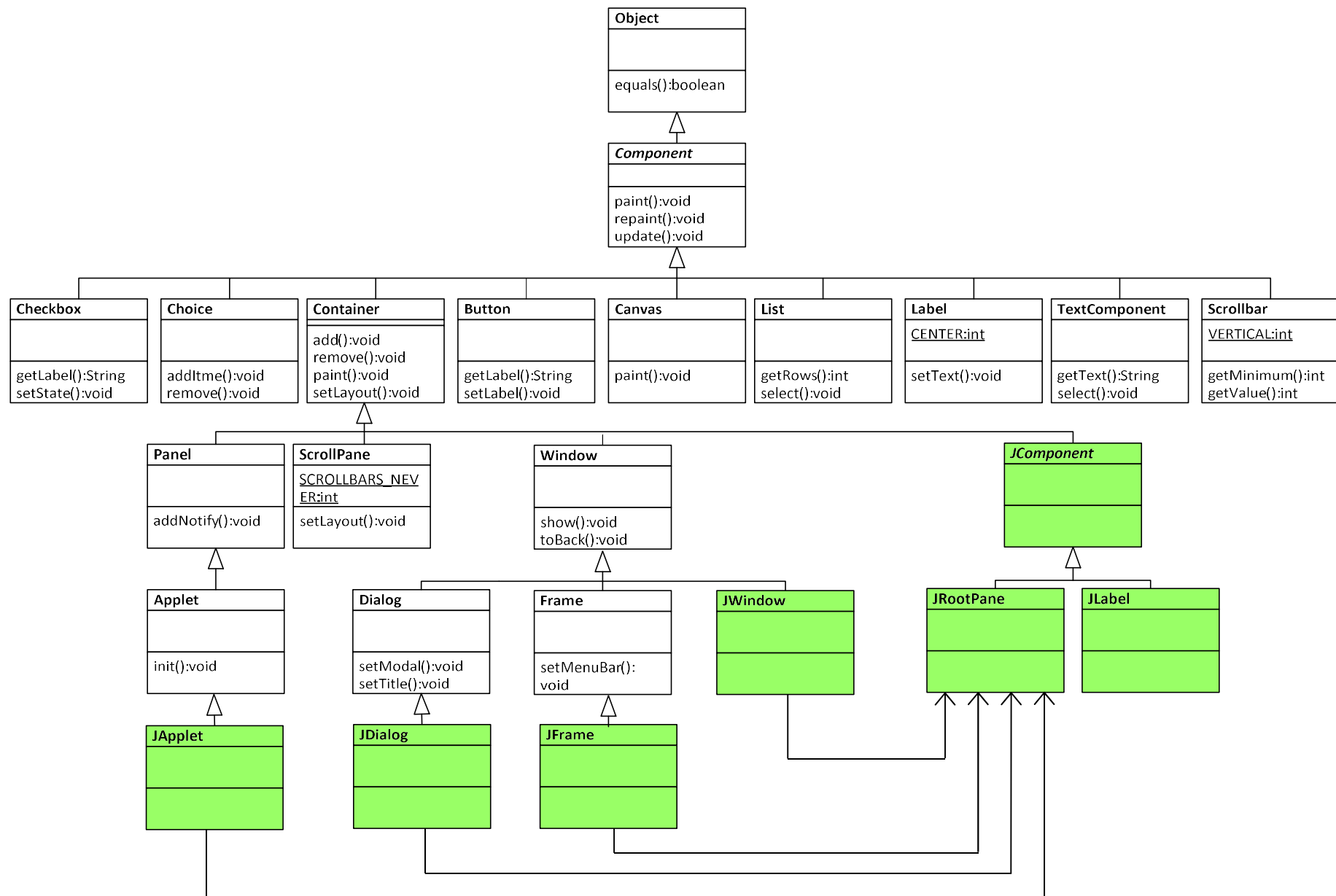


Swing – die Weiterentwicklung

- Bis und mit der Version Java 7 (1.7) wurde Swing sukzessive weiterentwickelt und verbessert.
 - Verbesserung der API, mehr Funktionalität.
 - Immer wieder neue Komponenten.
 - Nutzung des M[VC]-Prinzips (von Anfang an).
- Ein Meilenstein waren die (austauschbaren) Look&Feels, mit welchen die Darstellung der Komponenten derjenigen der nativen GUI-Komponenten nachempfunden wurde.
 - Damit ist es auch heute noch möglich z.B. auf Windows den Motif-Style (Unix) zu aktivieren!
- Beispiel: NetBeans basiert vollständig auf Swing.



Swing-Klassen: Auszug, grüne Klassen



Swing – Aktueller Stand

- Swing ist derzeit die mit Abstand am meisten verwendete GUI-Technologie für Java-Applikationen.
- Vor wenigen Jahren wurde die Weiterentwicklung von Swing (zugunsten einer neuen GUI-Plattform JavaFX) eingestellt.
 - Swing wird aber noch aktiv gewartet.
- Das hat viele AnwenderInnen von Java stark verunsichert, da JavaFX relativ lange benötigte, bis es einen brauchbaren, produktionsreifen Stand erreichte.
 - Mit der Integration in Java 8 ist das nun aber gegeben.
- Konkret: Während bestehende Applikationen zu meist auf Swing basieren, ist für Neuentwicklungen in der Regel JavaFX klar vorzuziehen.

JavaFX

JavaFX

- Eigenwerbung: Oracle's neues GUI-Framework für plattformübergreifende „Rich Applications“.
- Implementiert wie Swing eigenständige Komponenten.
 - Komponenten wurden gegenüber Swing stark erweitert und die API ist mehr „Nutzergetrieben“, d.h. einfachere Nutzung.
- Vereinfachtes, stark modernisiertes Programmiermodell
 - Konzeptioneller, verwendet z.B. **Stage** und **Scenes**.
- Möglichkeit GUI's per FXML (➔ SceneBuilder) zu definieren.
 - Vergleichbar mit XUL und XAML.
 - Grosses Potential für externes Design, Ergonomie etc.
- Möglichkeit Darstellung mittels Stylesheets (CSS) anzupassen.
 - Ohne Codeänderung oder Neukompilierung.

Grundlagen von Java FX

- Java FX ist im Vergleich zu Swing ein echtes Framework, weil es den Aufbau eines GUIs deutlich mehr abstrahiert, vereinfacht und automatisiert.
- Konzept von **Stage**, **Scene**, **Node** und Scene Graph, stark vereinfacht:
 - **Stage** ist eine Analogie zum Frame (Window)
 - **Scene** ist eine Analogie zum (Root-)Panel (Content)
 - **Node** ist die Basisklasse für alle GUI-Elemente / Komponenten
 - Scene Graph ist eine (logische) hierarchische Datenstruktur welche alle Nodes enthält.
- JavaFX übernimmt das Rendering (Darstellung) des ganzen Scene Graph vollständig.
 - Erlaubt z.B. die sehr einfache Realisation von Animationen.

Java FX – FXML (und CSS)

- Ebenfalls herausragend an JavaFX ist, dass es damit (für Java) erstmals möglich ist, das Design der Visualisierung von der eigentlichen Programmierung vollständig zu trennen: FXML
- FXML (**FX-XML**) ist eine auf XML basierende Definition für User-Interfaces (vergleichbar mit XUL und XAML).

- Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<BorderPane prefHeight="100.0" prefWidth="200.0" ... >
  <top>
    <Button fx:id="btnOn" mnemonicParsing="false" text="On" BorderPane.alignment="CENTER" />
  </top>
  <bottom>
    <Button fx:id="btnOff" mnemonicParsing="false" text="Off" BorderPane.alignment="CENTER" />
  </bottom>
  <center>
    <Label fx:id="label" text="The switch is OFF." BorderPane.alignment="CENTER" />
  </center>
</BorderPane>
```



Java FX – Verbindung von FXML mit Code

- Die Verbindung zwischen FXML und dem Code (welcher z.B. die Events verarbeitet) geschieht über die «**fx:id**» - einfache **String**-Bezeichner die als Identifikation (ID) dienen.
 - Fluch und Segen: Tippfehler können sich stark auswirken!
- Programmierung wird vereinfacht, indem sogenannte
 - ➔ Dependency Injection verwendet wird, um die vom JavaFX-Framework aus der FXML-Beschreibung erzeugten GUI-Elemente in die Controller-Objekte zu setzen (Setter-Methoden).
 - **@FXML**-Annotationen für Attribute.

```
public final class SwitchGuiJavaFXMLController implements Initializable {  
    @FXML  
    private Button btnOn;           // fx:id="btnOn"  
    @FXML  
    private Button btnOff;          // fx:id="btnOff"  
    ...  
}
```

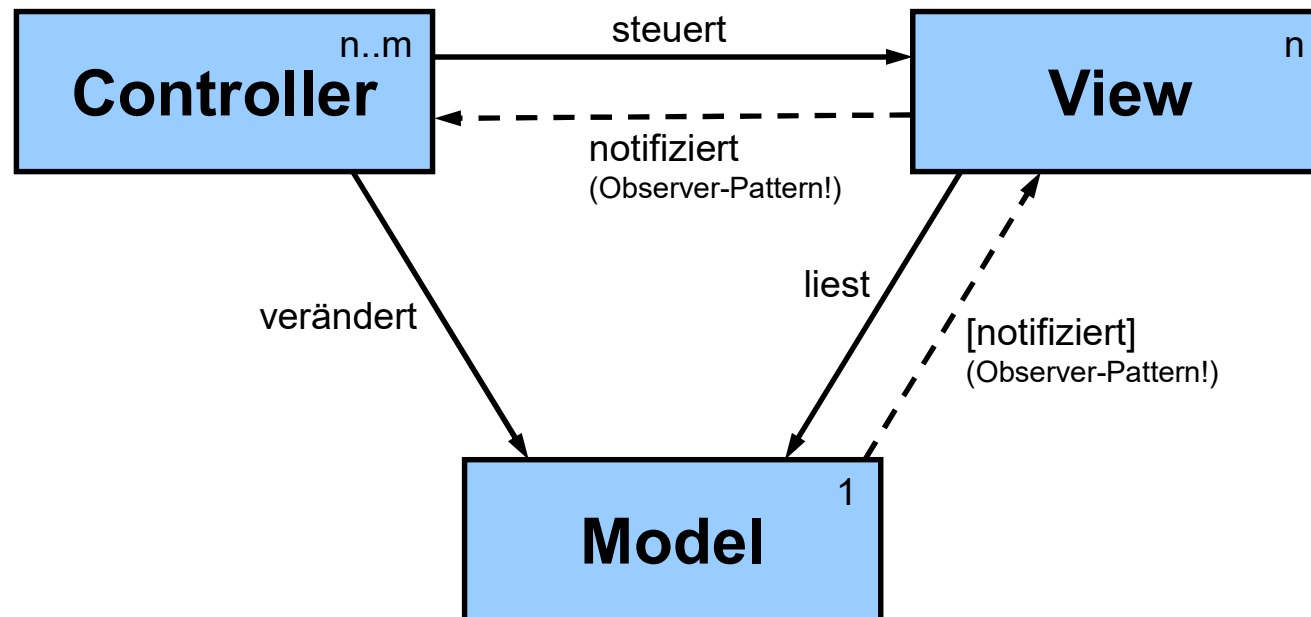
Model – View – Control (MVC)

Model – View – Control (MVC)

- Nach den Prinzipien SRP und SoC sollte man Aufgaben möglichst strikte voneinander Trennen, um lose Kopplung, hohe Kohäsion, eine bessere Wiederverwendung (und viele weitere Vorteile) zu erreichen.
- Bei der GUI-Programmierung kommen viele Dinge zusammen: Man steuert (kontrolliert) welche Daten (Modelle) auf welche Art wo und wann dargestellt werden.
- Gemäss SRP versucht man mindestens diese drei Teile möglichst gut voneinander zu trennen.
 - **Model** – Die eigentlichen Datenobjekte, z.B. eine Person.
 - **View** – eine GUI-Komponente welche die Daten anzeigt.
 - **Control** – eine (meist übergeordnete) Steuerung, welche das Ganze (bzw. mindestens M und V) koordiniert.

MVC - Grundidee

- Vereinfachte, konzeptionelle Darstellung (nicht UML):



- Die Pfeile zeigen in Richtung der Abhängigkeit:
 - **Model:** Ist völlig unabhängig und wird «**nur**» verwendet.
 - **View:** Verschiedene **Varianten** für das **selbe** Model möglich.
 - **Control:** Verbindet Modell und View, **steuert** und **koordiniert**.

MVC - Hintergrund

- MVC ist ein sehr fundamentales Konzept, das auf unterschiedlichen Abstraktionsebenen immer wieder eingesetzt werden kann.
 - Häufig auch hierarchisch verschachtelt.
- Es gibt nicht eine einzige, richtige Form und Implementation!
 - Wird je nach Situation, Technologie und Intention adaptiert.
- Es existieren verschiedenste Varianten und Verfeinerungen
 - Zum Beispiel: MVC2, MMVC, MVP etc.
- Allen gemeinsam: «Gewaltentrennung» zwischen dem Modell (Daten), deren Visualisierung (View) und deren Steuerung und Koordination (Control).
 - Modell ist beständig und unabhängig, die View abhängig vom Modell und der Controller ist meistens sehr stark spezialisiert.
- mehr dazu z.B. auch im Modul APPE (4. Semester)

Vorgehen: Programmieren oder GUI-Designer?

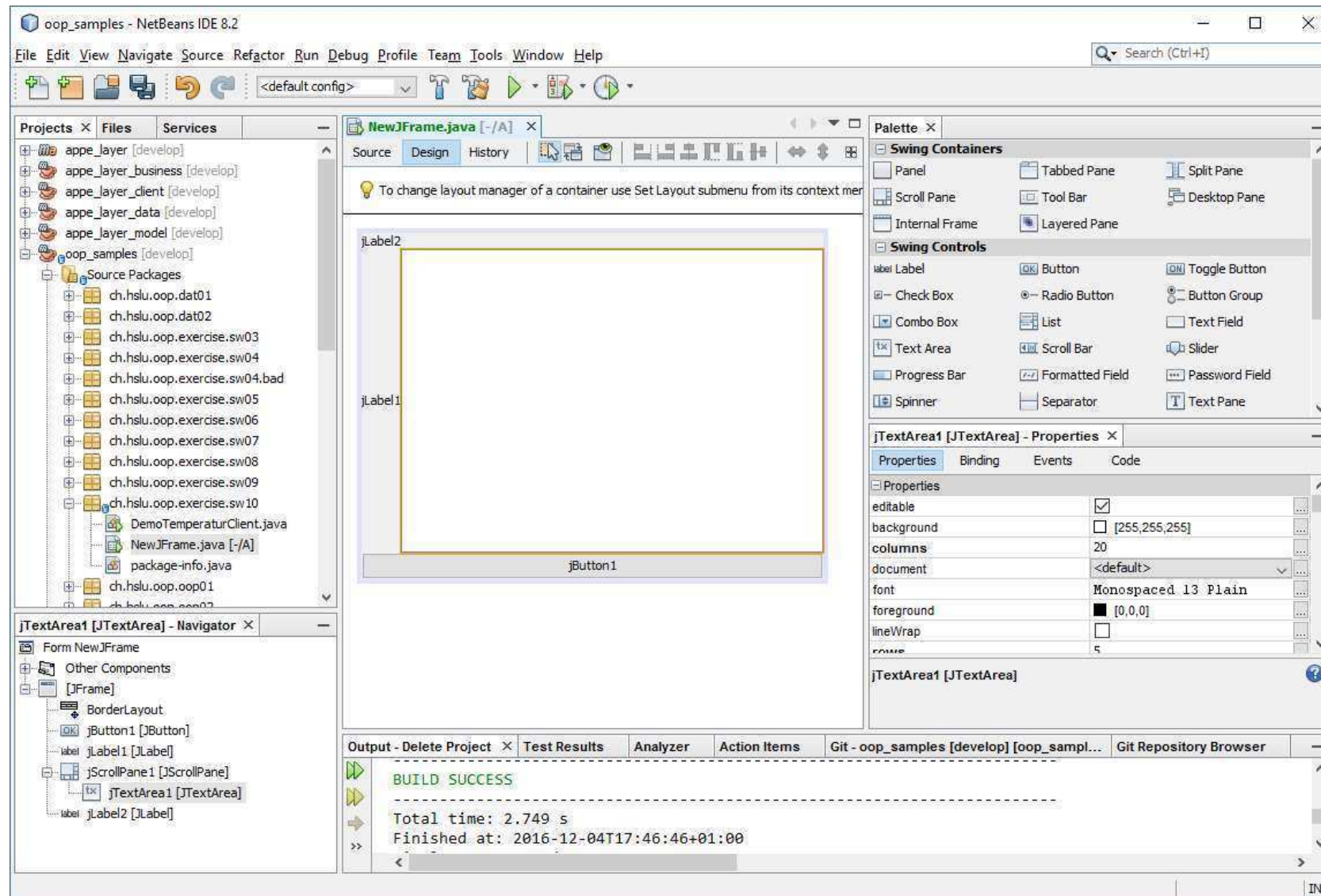
Variante 1: Programmierung

- Klingt ein bisschen nach «hardcore»:
Ein GUI von A bis Z zu Programmieren ist sehr aufwändig.
- Tatsächlich ist der Initialaufwand sicherlich höher.
 - Sinkt aber mit guter Vorbereitung: Planung, wie das GUI aufgebaut und strukturiert werden soll, ist ohnehin essentiell!
- Code ist in der Regel gut verständlich und effizient, es sind später sehr gezielte Änderungen und Korrekturen möglich.
- Hauptnachteil: «Nur» ProgrammierInnen können das GUI verändern, DesignerInnen sind in der Regel überfordert.
- Wichtiges Kriterium: Wie «dynamisch» (Häufigkeit) muss sich ein GUI an neue Anforderungen anpassen können?

Variante 2: GUI-Designer

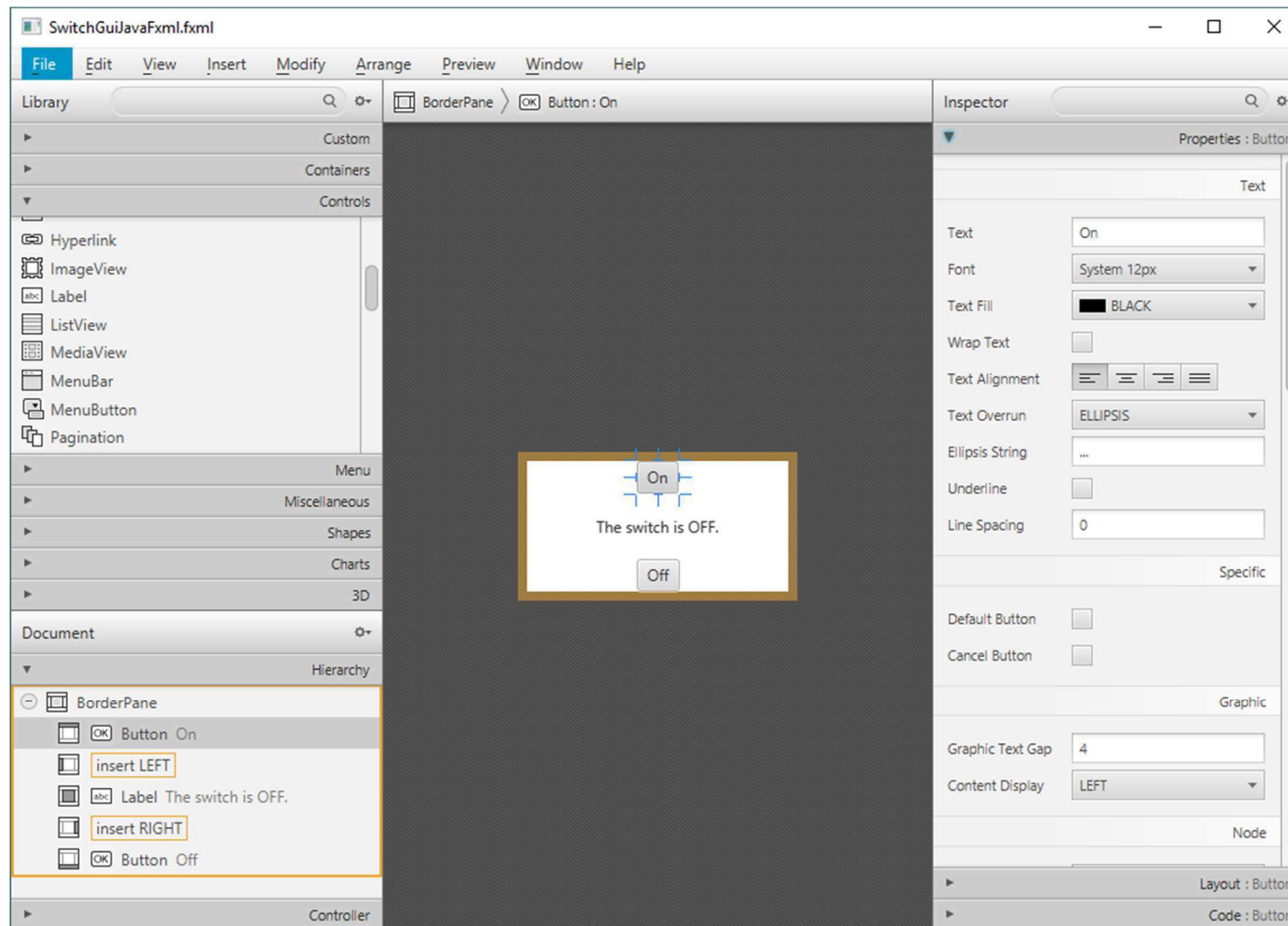
- Swing:
 - GUI-Designer welcher direkt Java-Code erzeugt, z.B. in NetBeans. Verleitet stark zu «Quick&Dirty»-GUIs (ohne Planung)
 - Code meist schwer verständlich, Anpassungen nur noch über GUI-Designer, Vorsicht vor Abhängigkeit zum Produkt / IDE.
 - JavaFX:
 - GUI-Definition technologisch über FXML weitgehend getrennt vom Code: Zwei (sinnvoll) getrennte Welten, erfüllt SRP.
 - Einzige Schwäche: Binding zw. FXML und Code per String-IDs.
 - Bilanz: Situation für Swing und JavaFX **nicht** identisch.
 - Für Code haben wir die Entwicklungsumgebung.
 - Für GUI-Design haben wir GUI-Editoren (z.B. SceneBuilder).
- ➔ JavaFX mit GUI-Designer klar im Vorteil?

GUI-Designer für Swing – Beispiel Netbeans



GUI-Designer für JavaFX – FXML-Authoring

- Gluon SceneBuilder - <http://gluonhq.com/labs/scene-builder/>



Empfehlungen zur GUI-Programmierung

- GUI-Programmierung ist aufwändig, darum ist eine gute Vorarbeit notwendig: GUI-Mockup für Design, Strukturierung, Anordnung, Ergonomie, Trennung der Aufgaben (MVC) → Viel Fleissarbeit!
- Neue Java-GUI's: Wenn möglich mit JavaFX (zukunftsgerichtet).
 - Die meisten bestehenden GUI's basieren aber auf Swing.
- Schwieriger Grundsatzentscheid:
Explizites Programmieren oder mit Hilfe von GUI-Designer?
 - Es gibt viele Argumente dafür und dagegen.
 - Nicht selten spaltet dieser Entscheid die Entwicklergemeinschaft.
 - Mittelfristig wird der GUI-Designer (im Zusammenhang mit «neuen» Technologien wie FXML und CSS) aber gewinnen.
- Ergonomie und Usability nicht vergessen!

Zusammenfassung

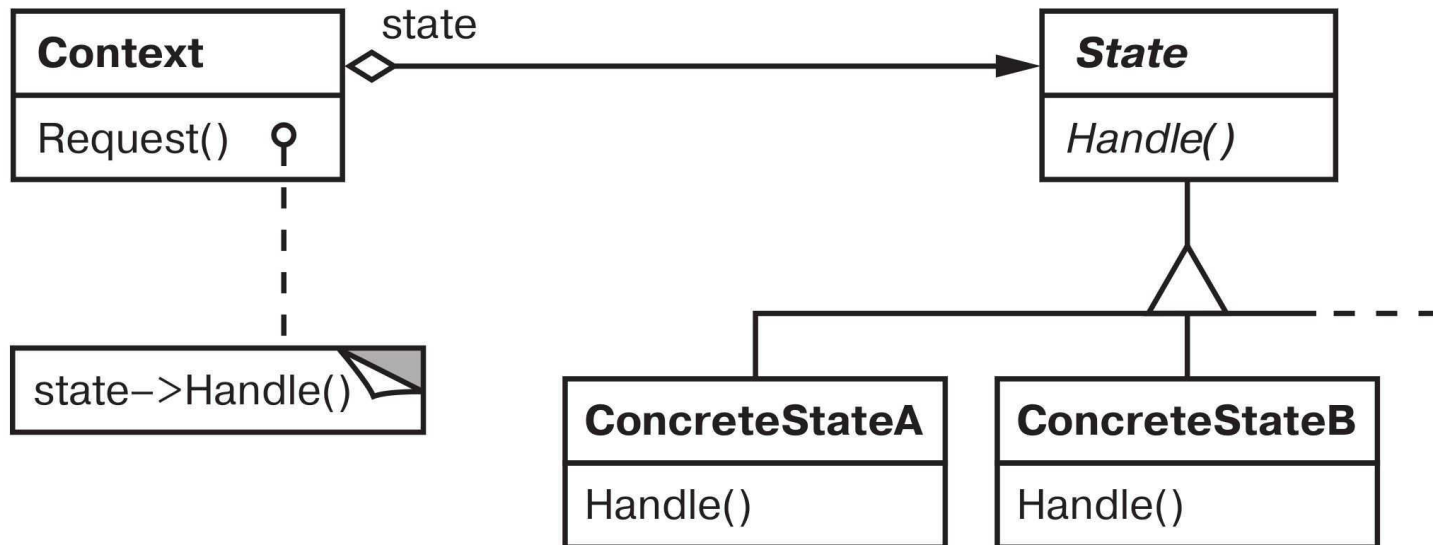
- GUI-Programmierung ist eine sehr umfangreiches Thema.
- Objektorientierte Strukturierung, Wiederverwendung von Komponenten und Containern, Composite-Pattern, hierarchisch.
- Ereignisorientierte Programmierung mit Observer-Pattern.
- (Möglichst) plattformunabhängige Implementation, Verwendung von Layout-Managern.
- Auswahl der Technologie/Framework: Swing oder JavaFX.
- Entwicklungsvorgehen: Immer zuerst eine Entwurf und Planung, erst dann Umsetzung – egal ob mit GUI-Designer oder native Programmierung.

Das State-Pattern

GoF 305

State-Pattern (1) – Theorie

State (305) Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class. (GoF)



State-Pattern (2) – Fallbeispiel Texteditor

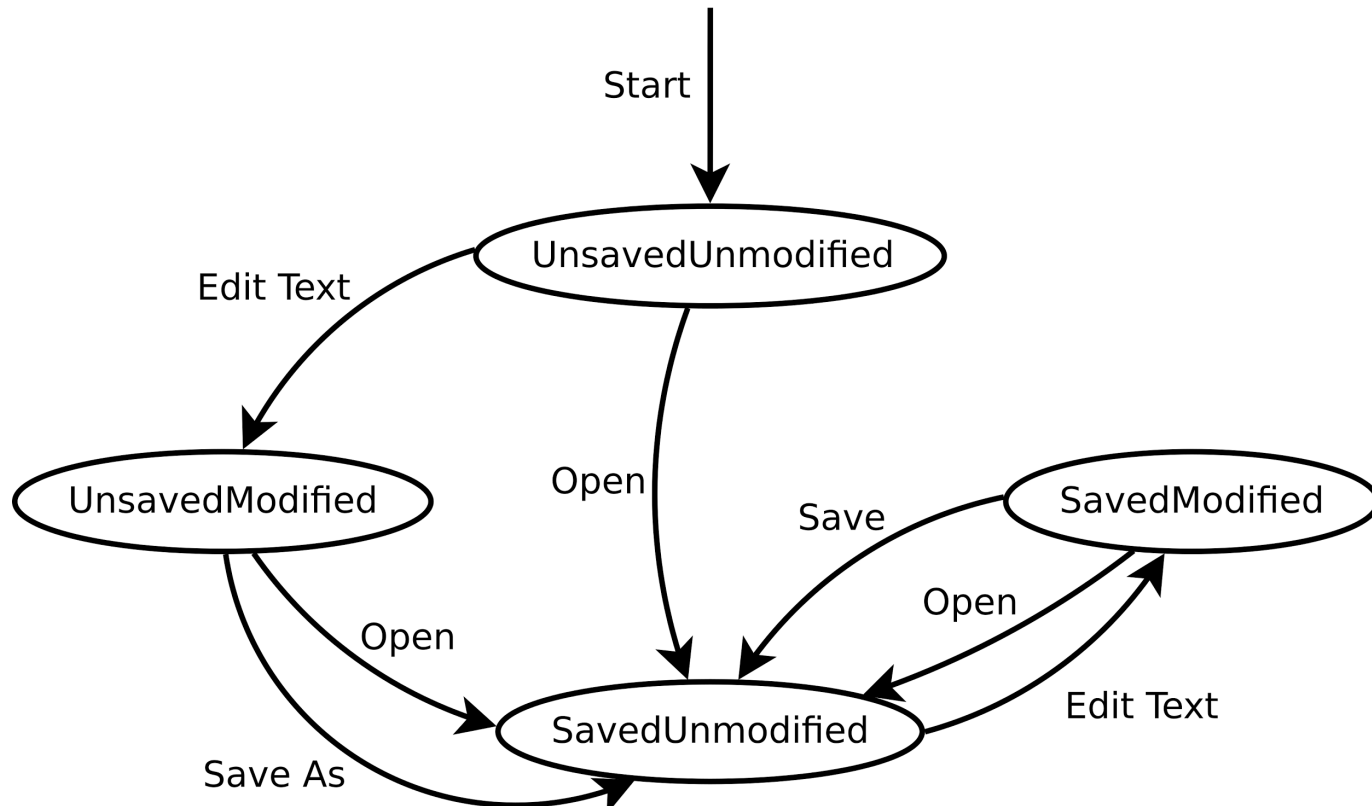
Der Inhalt eines Texteditors kann entweder modifiziert oder unmodifiziert sein. Er ist entweder als Datei abgespeichert oder nicht abgespeichert.

Wir unterscheiden nach zwei Kategorien (Modifikation und Speicherung) mit je zwei Zuständen und erhalten vier Klassen:

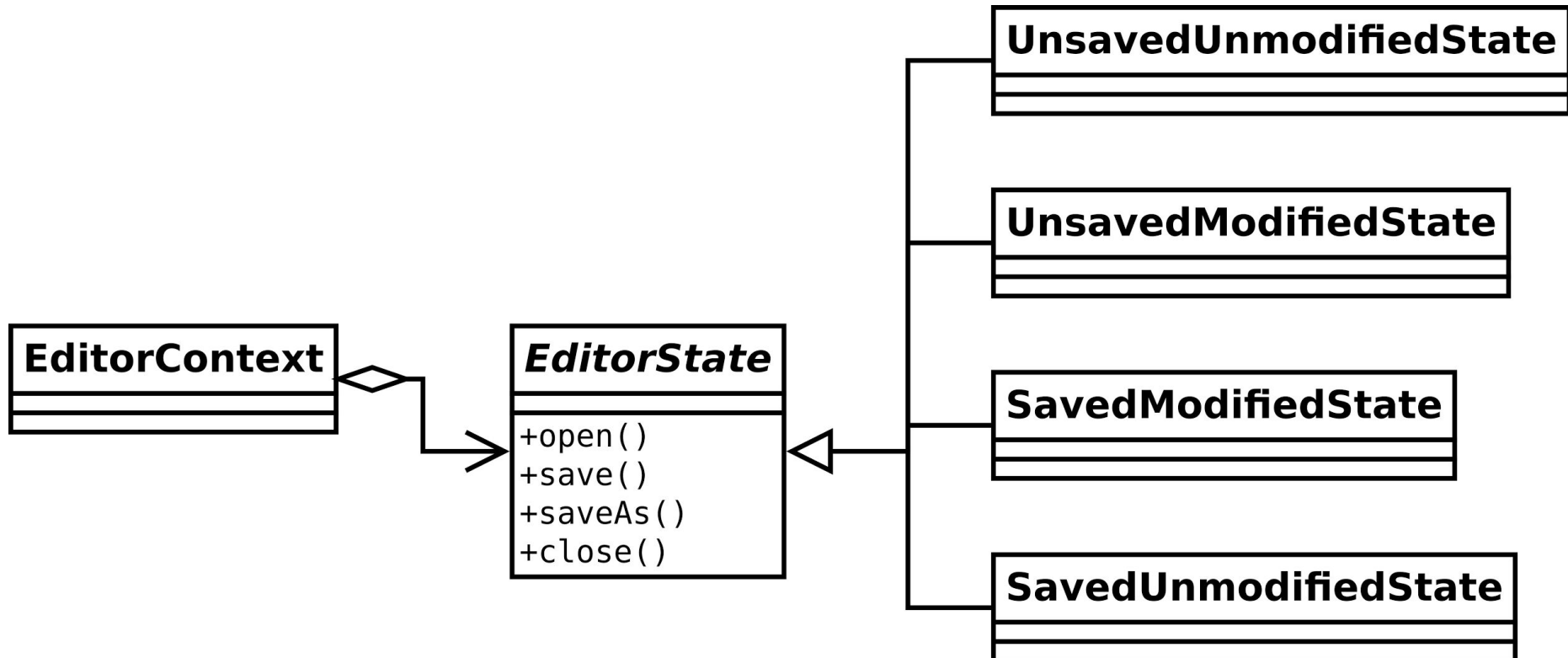
1. `UnsavedUnmodifiedState`
2. `UnsavedModifiedState`
3. `SavedUnmodifiedState`
4. `SavedModifiedState`

Frage: Was würde passieren, wenn eine weitere Kategorie dazu käme?

State-Pattern (3) – Endlicher Automat



State-Pattern (4) – Klassendiagramm



State-Pattern (5) – Implementierung

[Code]

State-Pattern (6) – Konsequenzen

- + Der Texteditor hat immer einen expliziten Status.
- + Das Verhalten jeder Aktion ist immer genau definiert.
- + Die Event-Handling-Methoden sind sehr schlank geworden.
- + EditorContext und EditorState sind lose über ein Interface gekoppelt...
 - ...greifen jedoch vielerorts und mit verschiedenen Methoden aufeinander zu.
 - Es wurden viele Klassen erstellt, die eine ähnliche Funktionalität haben.
 - Dadurch kommt es zu Code-Duplikation – oder zur Auslagerung von Code in “Utility”- oder “Helper”-Klassen (*code smell*).

Statt eines State-Interfaces könnte eine abstrakte Klasse implementiert werden, die gemeinsam genutzten Code beinhaltet.

Eine lose Kopplung ist schwer zu erreichen, wenn GUI-Funktionalität auf verschiedene Klassen verteilt wird.

State-Pattern (7) – Aufgabe 1

Lege einen neuen Branch (aufgabe-1) an. Implementiere dann den Menü-Eintrag *New* und die Operation *new()* im neuen Branch.

1. Welche neuen Statusübergänge ergeben sich dadurch?
2. Was ist zu tun, wenn der Benutzer Änderungen am Text vorgenommen hat und auf *New* klickt?
3. Welche Klassen müssen verändert und/oder erweitert werden?
4. Reflexion: Waren diese Änderungen einfach zu bewältigen?
Erleichtert oder erschwert das State-Pattern die Erweiterbarkeit?

Zusatzaufgabe: Was passiert, wenn das Fenster nicht über *Close* geschlossen wird? Muss das Verhalten erweitert werden? Wenn ja, wie?

State-Pattern (8) – Aufgabe 2

Lege einen neuen Branch (aufgabe-2) an. Ersetze im neuen Branch das Interface `EditorState` durch eine abstrakte Klasse.

1. Welcher Code der `EditorState`-Implementierungen (bzw. neu: Unterklassen) lässt sich in die abstrakte Klasse verschieben?
2. Wäre der Code von `DialogUtils` auch besser in der abstrakten Klasse aufgehoben?
3. Muss für diese Änderungen etwas an `EditorContext` oder an `TextEditor` verändert werden?
4. Reflexion: Wie hat sich diese Änderung auf den Code ausgewirkt? Was funktioniert hier besser: eine abstrakte Klasse oder ein Interface?

State-Pattern (9) – Refactoring

Wer glaubt, das State-Pattern führe zu unnötig kompliziertem und schlecht erweiterbarem Code, der kann ein Refactoring vornehmen:

1. Neuer Branch refactoring (ausgehend von master!)
2. Löschen der Interfaces `EditorContext` und `EditorState`
3. Handhabung der Operationen direkt in den Event-Handling-Methoden bzw. in neuen Zusatzmethoden der Klasse `TextEditor`
4. “Ausschlachten” und Löschen aller `EditorState`-Implementierungen

Ist der Code nun verständlicher?

Lassen sich der Menü-Eintrag *New* und die Operation `new()` vor oder nach dem Refactoring einfacher einfügen? (Versuche Aufgabe 1 auch im Branch refactoring zu lösen.)