Analyse und Modellierung von Informationssystemen

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2014/15

Singleton-Muster

Beispiel für die Kombination von Erzeugungsmustern

Das Singleton

Im einfachsten Fall wird ein Singleton über ein statisches Attribut realisiert:

```
public class Singleton1 {
    private static Singleton1 instance = null;

    private Singleton1() {}

    public static Singleton1 getInstance() {
        if (instance==null) {
            instance = new Singleton1();
        }
        return instance;
    }
}
```

Allerdings ist diese Variante nicht Thread-safe, da der zwischen der if-Abrage (im Wahr-Fall) und dem Erzeugen der Instanz ein kritischer Bereich liegt.

Thread-safe, Variante 1

```
public class Singleton2 {
    private static Singleton2 instance = null;

    private Singleton2() {}

    synchronized public static Singleton2 getInstance() {
        if (instance==null) {
            instance = new Singleton2();
        }
        return instance;
    }
}
```

Durch die synchronisierte Methode findet jedoch bei *jedem* Aufruf (auch wenn die Instanz schon längst existiert) eine Verriegelung statt, dies kostet Performance.

Thread-safe, Variante 2

```
public class Singleton3 {
    private static volatile Singleton3 instance = null;

    private Singleton3() {}

    public static Singleton3 getInstance() {
        if (instance==null) {
            synchronized (Singleton3.class) {
                if (instance==null) {
                    instance = new Singleton3();
                }
            }
        }
        return instance;
    }
}
```

Hier findet ein *Double-Checked Locking* statt, d. h. die Existenz der Instanz wird zweimal geprüft, nur wenn sie noch nicht existiert, findet die Verriegelung statt, danach nicht mehr.

Alternative Variante, Java-spezifisch

Die folgende Variante (*Initialization-on-Demand Holder Idiom* genannt), macht Gebrauch davon, dass innere Klassen in Java erst dann initialisiert werden, wenn sie tatsächlich referenziert, also benutzt werden:

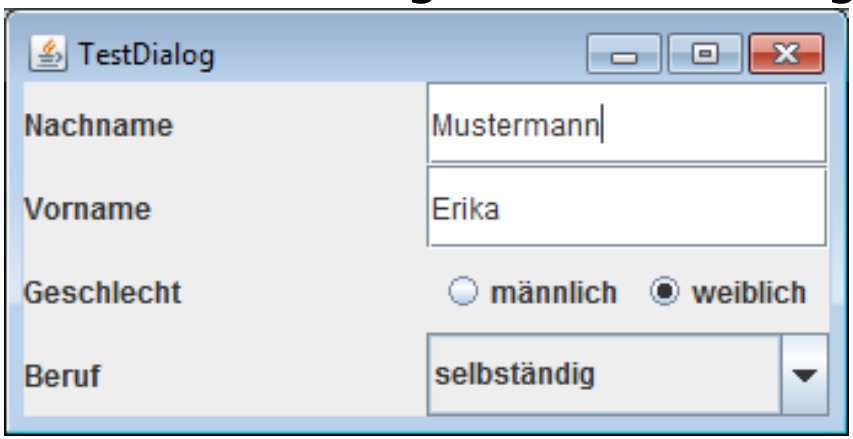
```
public class Singleton4 {
    private Singleton4() {}

    private static class Helper {
        private static final Singleton4 instance =
                new Singleton4();
    }

    public static Singleton4 createInstance() {
        return Helper.instance;
    }
}
```

Konkretes Beispiel mit Erzeugungsmustern

Im folgenden wird eine Anwendung entwickelt, die aus einer externen Datenquelle (in diesem Fall eine HSQL-Datenbank oder eine XML-Datei) ein Dialog-Fenster (Swing) erzeugt:



Hierbei werden als Eingabeelemente Textfelder, Radiobuttons und Pulldown-Menüs unterstützt.
Alternativ soll eine HTML-Seite entstehen.

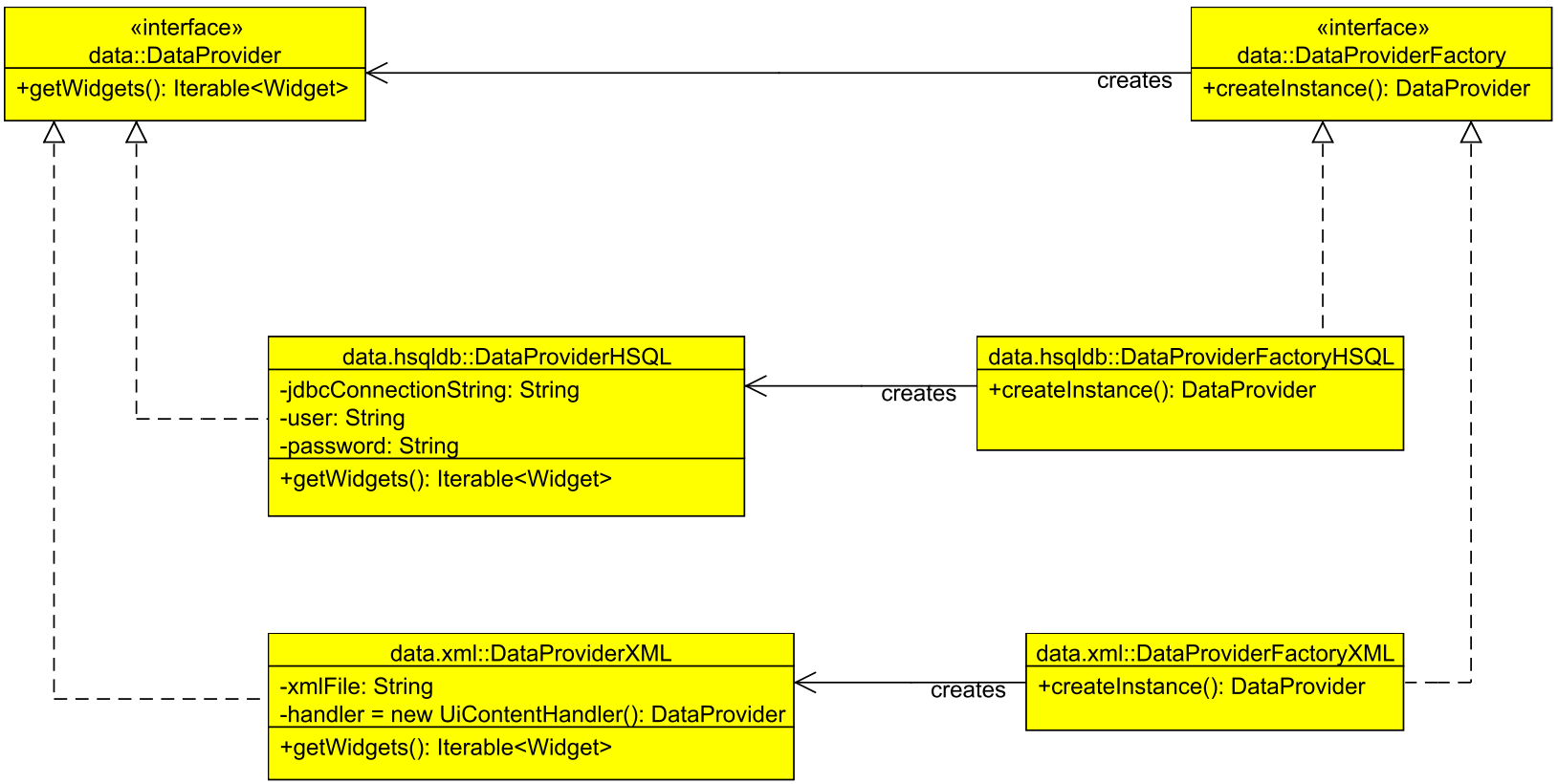
Dynamisches Konstruieren einer Oberfläche

Die wesentliche Aufgabe des Beispiels besteht darin, einen Dialog dynamisch aus einer Datenquelle aufzubauen.

Hierbei sollten die konkrete Datenquelle und die Präsentation des Dialogs weitgehend gekapselt sein. Damit soll erreicht werden, dass

- die Datenquelle für die Dialogelemente flexibel wählbar ist, z. B. eine Datenbank über JDBC oder JPA oder aus einer XML-Datei über SAX oder DOM,
- das Ausgabemedium des Dialogs leicht geändert werden kann, z. B. der Dialog als Swing-Applikation laufen kann oder durch ein HTML-Formular repräsentiert wird,
- wobei Datenquelle und Ausgabeformat voneinander unabhängig sind.

Teil 1: Bereitstellen der Datenquelle



Interfaces und konkrete Klassen

In dem Klassendiagramm sieht man, dass für die Datenquelle ein Interface `DataProvider` definiert wird, dass über die Methode `getWidgets()` eine abzählbare Menge von Widgets liefert, die im Dialog dargestellt werden. Die konkrete Implementierung findet sich dann in Klassen, die aus einer realen Datenquelle, z. B. einer HSQL-Datenbank lesen.

In der Anwendung selber wird die Datenquelle von einer Fabrik erzeugt, wobei diese selber wieder abstrakt ist, über das Interface `DataProviderFactory`.

```
DataProviderFactory factory = new ConcreteDataFactory();
DataProvider data = factory.createInstance();
```

Erzeugen der Widgets aus der HSQL-Datenbank

In der Klasse DataProviderHSQL:

```
public Iterable<Widget> getWidgets() {
    List<Widget> widgets = new ArrayList<Widget>();
    Class.forName("org.hsqldb.jdbc.JDBCDriver");
    Connection conn = DriverManager.getConnection(...);
    PreparedStatement widgetStmt = conn.prepareStatement(...);
    ResultSet widgetSet = widgetStmt.executeQuery();
    while (widgetSet.next()) {
        WidgetImpl widget = new WidgetImpl();
        int id = widgetSet.getInt("id");
        widget.setLabel(widgetSet.getString("label"));
        widget.setType(
            Widget.Types.valueOf(widgetSet.getString("typename"))
        );
        ...
        widgets.add(widget);
    }
    conn.close();
    return widgets;
}
```

Datenquelle XML-Datei

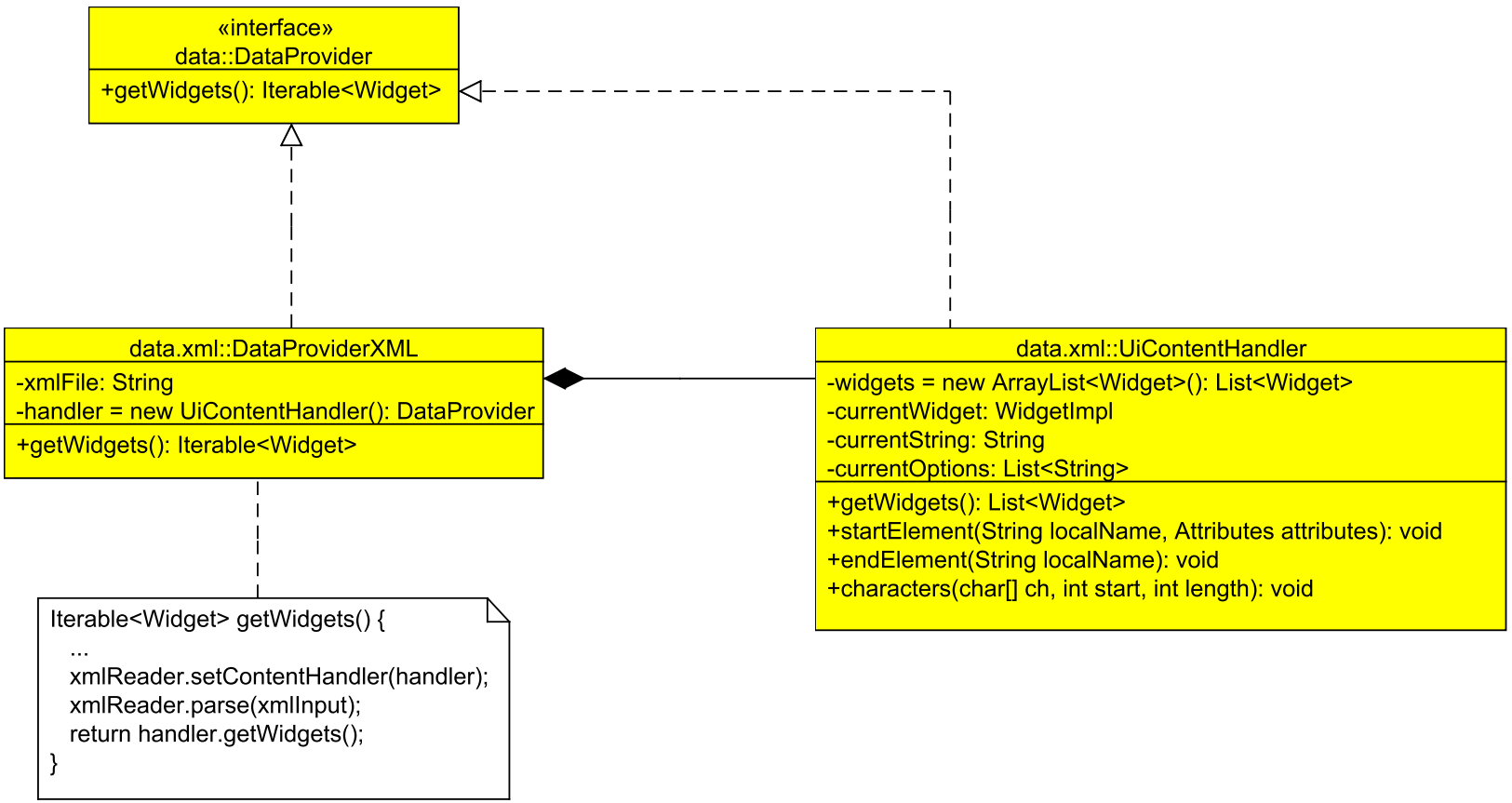
Alternativ können die Widget-Definitionen aus einer XML-Datei eingelesen werden.

Hierfür wird die XML-Datei über SAX (Simpe API for XML) geparst. Hierbei wird eine ContentHandler-Klasse verwendet, die Callbacks für die Ereignisse Start und Ende eine XML-Tags und Text zur Verfügung stellt.

In der Realisierung implementiert `DataProviderXML` zwar das Interface `DataProvider`, das eigentliche Erzeugen der Widget-Iteration wird aber von der verwalteten Instanz von `UiContentHandler` erledigt, die ebenfalls `DataProvider` implementiert.

Eine solche Konstruktion findet man in diversen Entwurfsmustern, z. B. im Strategiemuster und im Dekorierer.

Delegation an den ContentHandler



Parsen der XML-Datei

In der Klasse `DataProviderXML`:

```
public Iterable<Widget> getWidgets() {
    XMLReader reader = XMLReaderFactory.createXMLReader();
    FileReader input = new FileReader(xmlFile);
    InputSource source = new InputSource(input);
    reader.setContentHandler((ContentHandler) handler);
    reader.parse(source);
    input.close();
    return handler.getWidgets();
}
```

Das eigentliche Erzeugen der Widget-Liste erledigt `handler`, der von `ContentHandler` erbt, aber auch `DataProvider` implementiert.

Der Handler

In `UiContentHandler extends ContentHandler implements DataProvider`:

```
private List<Widget> widgets = ...;

public void startElement(String localName, Attributes attributes) {
    if (localName.equals("widget")) {
        currentWidget = new WidgetImpl();
        currentWidget.setType(
            Widget.Types.valueOf(attributes.getValue("type")));
    } else if (...) { ... }
}

public void endElement(String localName) {
    if (localName.equals("widget")) {
        widgets.add(currentWidget);
    } else if (...) { ... }
}

public Iterable<Widget> getWidgets() {
    return widgets;
}
```

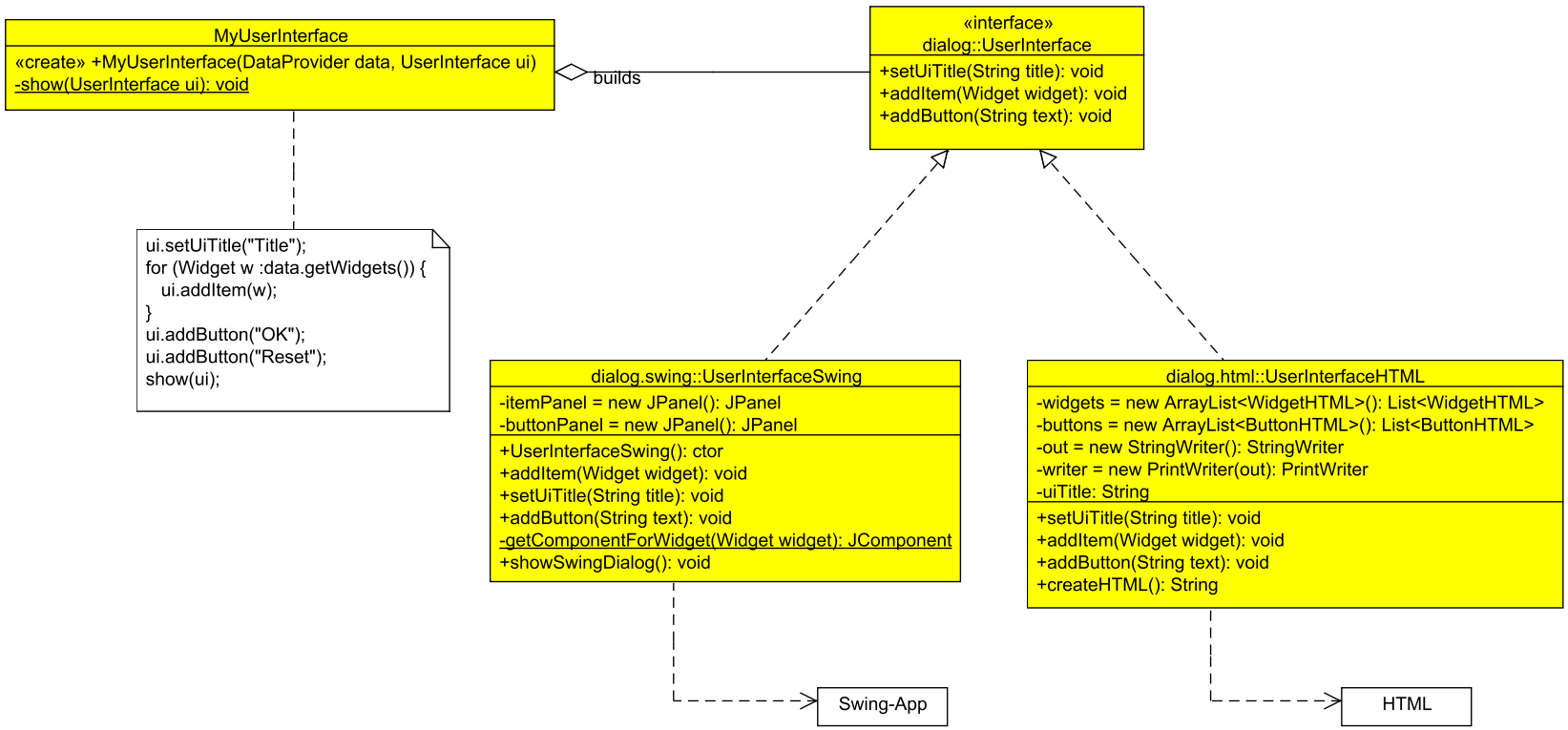
Teil 2: Generieren des Dialogs

Im folgenden Schritt soll aus der aus einer Datenquelle ausgelesenen Liste von Widgets die eigentlich optische Repräsentation des Dialogs erfolgen, also eine Datenmenge in eine sichtbare Form konvertiert werden.

Eine solche Aufgabe wird in den Entwurfsmustern der GoF durch den Erbauer (*Builder*) realisiert, wobei ein Interface (abstrakte) Methoden zum schrittweisen Aufbau eines gewünschten Ergebnisses aus einer Datenquelle definiert. Die konkreten Implementierungen zur realen Darstellung des Ergebnisses werden dann in Klassen zur Verfügung gestellt, die das Erbauer-Interface implementieren.

Das Entwurfsmuster besteht daher aus dem Erbauer-Interface, den konkreten Erbauern und einem Direktor, der oft in einer Iteration aus den Daten das Resultat konstruiert.

Das Konstruieren des Dialogs



Die Klasse UserInterfaceSwing

```
public class UserInterfaceSwing extends JFrame implements UserInterface {
   JPanel itemPanel = new JPanel();
   JPanel buttonPanel = new JPanel();

   public UserInterfaceSwing() {
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLayout(new BoxLayout(getContentPane(), BoxLayout.PAGE_AXIS));
      itemPanel.setLayout(new GridLayout(0, 2));
      add(itemPanel);
      buttonPanel.setLayout(
         new BoxLayout(buttonPanel, BoxLayout.LINE_AXIS));
      buttonPanel.add(Box.createHorizontalGlue());
      add(buttonPanel);
   }

   @Override
   public void setUiTitle(String title) {
      setTitle(title);
   }
```

Die Klasse UserInterfaceSwing (Forts.)

```
@Override
public void addButton(String text) {
   buttonPanel.add(new JButton(text));
}

@Override
public void addItem(Widget widget) {
   itemPanel.add(new JLabel(widget.getLabel()));
   itemPanel.add(getComponentForWidget(widget));
}

public void showSwingDialog() {
   pack();
   setVisible(true);
}
```

Die Klasse UserInterfaceSwing (Forts.)

```
private static JComponent getComponentForWidget(Widget widget) {
   JComponent c = null;
   switch (widget.getType()) {
   case TEXT:
      c = new JTextField();
      break;
   case RADIO:
      c = new JPanel();
      c.setLayout(new BoxLayout(c, BoxLayout.LINE_AXIS));
      ButtonGroup group = new ButtonGroup();
      for (String option : widget.getOptions()) {
         JRadioButton b = new JRadioButton(option);
         c.add(b);
         group.add(b);
      }
      break;
```

Die Klasse UserInterfaceSwing (Forts.)

```
        case PULLDOWN:
            JComboBox<String> combo = new JComboBox<String>();
            for (String option : widget.getOptions()) {
                combo.addItem(option);
            }
            c = combo;
            break;
        }
        return c;
    }
}
```

Generieren der HTML-Datei

```
public class UserInterfaceHTML implements UserInterface {
   List<WidgetHTML> widgets = new ArrayList<WidgetHTML>();

   public void addItem(Widget widget) {
      widgets.add(new WidgetHTML(widget));
   }
   public String createHTML() {
      StringWriter out = new StringWriter();
      PrintWriter writer = new PrintWriter(out);
      ...
      writer.println("<table>");
      for (WidgetHTML widget : widgets) {
         writer.println("<tr>");
         writer.format("<td>%s</td><td>%s</td>\n",
            widget.getLabel(),
            widget.createHTMLElement());
         writer.println("</tr>");
      }
      writer.println("<table>");
      ...
      return out.toString();
   }
}
```

Dekorieren von Klassen

Die Klasse `UserInterfaceHTML` verwendet eine Klasse `WidgetHTML`, die das Interface `Widget` implementiert. Tatsächlich realisiert `WidgetHTML` jedoch die im Interface definierten Methoden nicht selber, sondern verwaltet intern eine Instanz eines Widgets, an die die entsprechenden Methodenaufrufe weitergeleitet werden.

Die Klasse `WidgetHTML` fügt eine neue Methode `createHTMLElement()` hinzu, die den HTML-Code des Formelementes für das darunterliegende Widget generiert. Dieses Hinzufügen neuer Funktionalität zu einem Objekt wird in dem Erzeugungmuster Dekorierer beschrieben.

Schaubild des Dekorierers