Nils Asmussen

Die Architektur des Java Web-Frameworks Apache Wicket

21. Dezember 2011

Technischer Bericht Nr. 2
des Instituts für SoftwareArchitektur

Inhaltsverzeichnis

1 Einleitung **2**

2 Beschreibung der Architektur **3**
2.1 Komponenten . . . 3
2.1.1 Grundlagen . . . 3
2.1.2 Eigene Komponenten . . . 4
2.1.3 Markup-Vererbung . . . 5
2.2 Modelle . . . 5
2.2.1 Grundlagen . . . 6
2.2.2 `PropertyModel` . . . 6
2.2.3 `CompoundPropertyModel` . . . 7
2.2.4 `LoadableDetachableModel` . . . 7
2.3 Events . . . 8
2.4 Behaviors . . . 8
2.5 Formulare . . . 9
2.5.1 Konvertierung . . . 9
2.5.2 Validierung . . . 9
2.5.3 Beispiel . . . 9
2.6 Ressourcen . . . 10
2.7 Anfrage-Bearbeitung . . . 11

3 Analyse der Architektur **13**
3.1 Komponenten . . . 13
3.2 Modelle . . . 14
3.3 Events . . . 15
3.4 Behaviors . . . 16
3.5 Ressourcen . . . 16
3.6 Anfrage-Bearbeitung . . . 17

4 Fazit **18**

Literaturverzeichnis **19**

1 Einleitung

Als Tim Berners-Lee im Jahre 1989 das World Wide Web kreierte [4], konnte sich vermutlich niemand vorstellen, dass einmal komplexe Anwendungen mit Hilfe dieser Technologie realisiert werden würden. Mittlerweile sind zu diesem Zweck unzählige Frameworks entstanden, die sich grob in zwei Kategorien einteilen lassen: Die einen sind „Request-Response-orientiert“ und daher sehr nah an HTTP und die anderen sind „Komponenten-orientiert“ und versuchen die Details von HTTP eher vor dem Anwendungsentwickler zu verstecken. Das Framework Apache Wicket zählt zu der zweiten Kategorie. Das Projekt wurde im Jahre 2004 von Jonathan Locke und Miko Matsumura ins Leben gerufen und ist als Open-Source unter der Apache Lizenz verfügbar [1]. Ziel ist es den Entwickler so weit wie möglich von der Funktionsweise des HTTP-Protokolls abzuschirmen, d.h. ein Programmiermodell zu bieten, dass eine zustandsbehaftete Schicht über das zustandslose HTTP legt [5, S. 10]. Einfachheit ist dabei ein Kernanliegen von Wicket. Denn Wicket ist „just Java, just HTML, and meaningful abstractions“ [5, S. 11]. Bei den Abstraktionen hat Wicket vor allem die einfache Wiederverwendbarkeit im Auge. Auf den folgenden Seiten beschreibt die vorliegende Arbeit die Architektur von Apache Wicket in der Version 1.5 und analysiert deren Konsequenzen.

2 Beschreibung der Architektur

Die zentralen Konzepte in Wicket sind Komponenten, Modelle, Behaviors und Events. Dabei ist die Grundidee, dass Webanwendungen durch Komponenten zusammengebaut werden, diese ein Modell für den Zugriff auf die Daten besitzen und mit Behaviors angereichert werden können. Zudem lösen einige Komponenten wie Links oder Formulare Events aus, auf die der Entwickler reagieren kann. Daher gestaltet sich die Entwicklung mit Wicket ähnlich der mit Java-Swing.

2.1 Komponenten

Der wichtigste Bestandteil von Wicket sind wohl die Komponenten, so dass mit ihnen begonnen werden soll.

2.1.1 Grundlagen

Komponenten sind Objekte von `Component` oder davon abgeleiteten Klassen. Sie werden in zwei Kategorien eingeteilt:

1. `MarkupContainer`: Komponenten, die andere Komponenten enthalten können.

2. `WebComponents`: Komponenten, die keine anderen enthalten.

Zu ersteren zählen z. B. `WebPage`, `Panel` und `Form`. Als Beispiele für `WebComponents` lassen sich `Label` und `Image` nennen. Die Wurzel dieses Komponentenbaums bildet stets ein Objekt von `WebPage`, welches - wie der Name schon andeutet - eine gesamte Webseite repräsentiert, die Knoten werden durch `MarkupContainer` besetzt und die Blätter durch `WebComponents`.

Jede Komponente hat eine ID, welche sie eindeutig innerhalb ihrer Ebene im Baum identifiziert. Des Weiteren besitzen einige Komponenten eine dazugehörige Markupdatei (typischerweise enthält sie HTML-Code). Dazu zählen z. B. `WebPage` und `Panel`. Die Markupdatei legt also das Design der Komponente fest. In ihnen wird durch spezielle Attribute die Verbindung zwischen HTML-Elementen und Komponenten hergestellt. Folgendes Beispiel veranschaulicht diesen Zusammenhang:

```
1  public class AddressPage extends WebPage {
2    public AddressPage() {
3      WebMarkupContainer address = new WebMarkupContainer("address");
4      address.add(new Label("street", "Wiesenstraße"));
5      address.add(new Label("number", "14"));
6      address.add(new Label("zipcode", "35390"));
7      address.add(new Label("city", "Gießen"));
8      add(address);
9    }
10 }
```

Listing 2.1: Der Java-Code der Page

```
<html>
<head>
  <title>Hello Page</title>
</head>
<body>

<div wicket:id="address">
  <span wicket:id="street">Musterstraße</span> <span wicket:id="number">5</span><br />
  <span wicket:id="zipcode">12345</span> <span wicket:id="city">Musterstadt</span>
</div>

</body>
</html>
```

Listing 2.2: Der dazugehörige HTML-Code (AddressPage.html)

Mit der Methode `MarkupContainer#add` werden dabei Komponenten andere Komponenten hinzugefügt. Das erste Argument von `WebMarkupContainer` und `Label` entspricht jeweils deren ID. Diese wird im Markup über `wicket:id="..."` referenziert. Wenn Wicket die Seite rendern soll, ersetzt `Label` den Inhalt des ihm zugeordneten Elements mit dem festgelegten Inhalt. Der `WebMarkupContainer` ist lediglich ein Behälter für Komponenten[1] und ändert daher nichts am Markup. Nach dem Rendervorgang sieht der Inhalt des Body-Elements somit folgendermaßen aus[2]:

```
<div wicket:id="address">
  <span wicket:id="street">Wiesenstraße</span> <span wicket:id="number">14</span><br />
  <span wicket:id="zipcode">35390</span> <span wicket:id="city">Gießen</span>
</div>
```

Listing 2.3: Ergebnis des Rendervorgangs

Wicket verlangt dabei, dass der im Java-Code produzierte Komponentenbaum stets mit dem durch die `wicket:id`-Attribute im Markup definierten Baum übereinstimmt. Das bedeutet, man kann zwar innerhalb einer Ebene im Baum Komponenten hin- und herschieben, kann jedoch z. B. nicht eine Komponente im Markup einer anderen Elternkomponente zuweisen. Elemente im Markup ohne `wicket:id`, wie das `br`-Element im Beispiel, können natürlich beliebig hinzugefügt, verschoben oder entfernt werden.

2.1.2 Eigene Komponenten

Ein wichtiges Anliegen von Wicket ist die einfache Wiederverwendbarkeit von Komponenten. Dies können sowohl fachlich motivierte Komponenten wie Formulare zur Eingabe einer Adresse sein oder allgemein verwendbare, d. h. technisch motivierte Komponenten wie ein Textfeld mit angehängtem „Date-Picker“ zur Eingabe/Auswahl eines Datums [6, S. 121,122]. Um aus den Bestandteilen des vorherigen Beispiels eine eigene Komponente zu bauen, genügt es eine Klasse von `Panel` abzuleiten, die `Label`-Objekte in deren Konstruktor zu sich selbst anstatt zum `WebMarkupContainer` hinzuzufügen und die Elemente im Markup des Panels mit `<wicket:panel>` zu umschließen:

```
public class AddressPanel extends Panel {
  public AddressPanel(String id) {
```

[1] Damit lassen sich z. B. mehrere Komponenten gleichzeitig ausblenden oder via AJAX aktualisieren.

[2] Natürlich lassen sich die `wicket:id`-Attribute auch herausnehmen. Dies ist im Produktionsmodus empfehlenswert um validen HTML-Code zu produzieren.

```java
    super(id);
    add(new Label("street", "Wiesenstraße"));
    add(new Label("number", "14"));
    add(new Label("zipcode", "35390"));
    add(new Label("city", "Gießen"));
  }
}
```

Listing 2.4: Der Java-Code des Panels

```html
<wicket:panel>
<span wicket:id="street">Musterstraße</span> <span wicket:id="number">5</span><br />
<span wicket:id="zipcode">12345</span> <span wicket:id="city">Musterstadt</span>
</wicket:panel>
```

Listing 2.5: Der dazugehörige HTML-Code (AddressPanel.html)

Anschließend kann dieser Panel auf sehr einfache Weise einem beliebigen `MarkupContainer` hinzugefügt werden:

```java
add(new AddressPanel("address"));
```

```html
<div wicket:id="address"></div>
```

2.1.3 Markup-Vererbung

Das Konzept der „Markup-Vererbung" überträgt den aus objektorientierten Sprachen bekannten Vererbungsmechanismus auf Markup. Auf diese Weise wird das Design durch zwei Markupdateien bestimmt; eine für die Oberklasse und eine für die Unterklasse. In der Markupdatei der Oberklasse kann die Stelle, an die das Markup der Unterklasse eingefügt werden soll, mittels des Platzhalters `<wicket:child/>` festgelegt werden. Die Abbildung 2.1 auf der nächsten Seite verdeutlich den dadurch entstehenden Zusammenhang anhand eines Beispiels. Auf der linken Seite ist die Ansicht im Browser dargestellt, welche durch die Markupdateien der einzelnen beteiligten Komponenten bestimmt wird. Der gelbe Bereich wird durch die Oberklasse bestimmt und der blaue Bereich durch die Unterklasse. Der mittlere Teil der Abbildung zeigt das dazugehörige Klassendiagramm. D. h. die Klasse `BasePage` leitet von `WebPage` ab und ist für den gelben Bereich der Seite verantwortlich. So legt sie das Markup für diesen Bereich fest und fügt die Komponenten (TextField und Panel) in den Komponenten-Baum ein. Davon abgeleitet existiert die Klasse `SomePage`, welche für den blauen Bereich zuständig ist. Die rechte Seite der Abbildung zeigt das Objektdiagramm, d.h. die Situation zur Laufzeit der Anwendung. Dies soll noch einmal verdeutlichen, dass es zur Laufzeit nur ein Objekt von `SomePage` gibt, welches die gesamte Seite im Browser repräsentiert.

2.2 Modelle

Modelle stellen die Verbindung zwischen den Komponenten und den Domänenobjekten der Anwendung her. Dazu besitzt jede Komponente ein Modell. Entscheidend ist, dass weder die Komponente über die angebundenen Domänenobjekte und deren Quelle Kenntnis besitzt noch umgekehrt die Domänenobjekte über ihre Darstellung Bescheid wissen [6, S. 185].

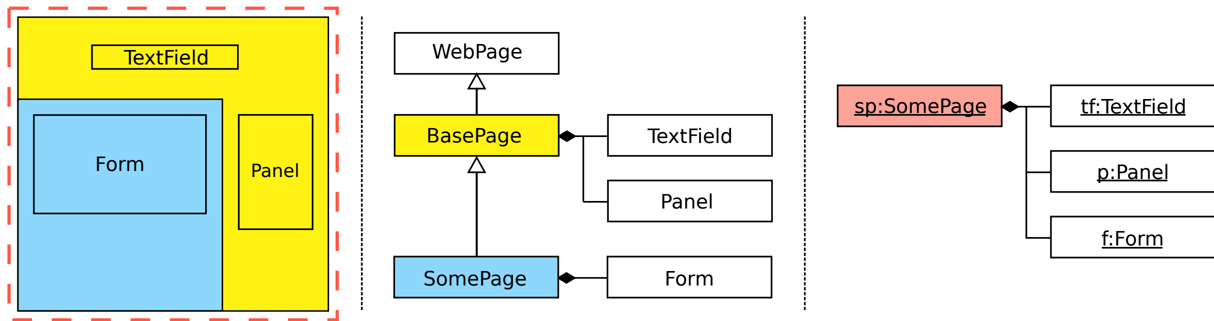

Abbildung 2.1: Verschiedene Sichten auf die Markup-Vererbung

2.2.1 Grundlagen

Technisch gesehen ist ein Modell eine Implementierung der Schnittstelle `IModel`, welche folgendermaßen aussieht:

```
1 public interface IModel<T> extends IDetachable {
2   T getObject();
3   void setObject(final T object);
4 }
```

Listing 2.6: Die Schnittstelle IModel

Ein Modell enthält also ein Objekt beliebigen Typs und bietet Möglichkeiten das Objekt zu bekommen und es auszutauschen. Wicket bringt bereits einige fertige Implementierungen dieser Schnittstelle mit. Die einfachste ist `Model`, welche lediglich einen Wrapper für ein beliebiges Objekt darstellt. Einige weitere wichtige und interessante Implementierungen werden in den folgenden Abschnitten genauer vorgestellt.

2.2.2 `PropertyModel`

Das `PropertyModel` dient zum Zugriff auf ein einzelnes Attribut eines Domänenobjektes anhand einer sogenannten Property-Expression. Auf diese Weise kann nicht nur einmalig der aktuelle Wert z. B. an `Model` für die Darstellung gegeben werden, sondern es wird stets direkt auf das Attribut des Domänenobjektes zugegriffen; lesend und auch schreibend, sofern erforderlich. Angenommen es gäbe eine Java-Bean namens `Address` mit entsprechenden Attributen, so könnte das `AddressPanel` aus dem vorherigen Beispiel das `PropertyModel` wie folgt einsetzen:

```
1 public AddressPanel(String id, Address addr) {
2   super(id);
3   add(new Label("street", new PropertyModel<String>(addr, "street")));
4   add(new Label("number", new PropertyModel<String>(addr, "number")));
5   add(new Label("zipcode", new PropertyModel<Integer>(addr, "zipcode")));
6   add(new Label("city", new PropertyModel<String>(addr, "city")));
7 }
```

Listing 2.7: Beispiel des `PropertyModels`

Auf diese Weise werden einzelne Attribute von `addr` an die `Labels` gebunden, wobei `"street"` für einen Aufruf von `addr.getStreet()` sorgt. Neben diesen einfachen Property-Expressions sind auch Zugriffe auf geschachelte Objekte (`"address.zipcode"`) und Arrays (`"array[2]"`) möglich.

2.2.3 `CompoundPropertyModel`

Da es häufig vorkommt, dass eine Komponente auf sämtliche Attribute eines Domänenobjektes zugreift, existiert die `IModel`-Implementierung `CompoundPropertyModel`. Diese ist eine Weiterführung des `PropertyModels` und ist besonders dann nützlich, wenn eine Komponente sozusagen ein Domänenobjekt repräsentiert. Z. B. ein Registrierungsformular mit diversen Angaben zu dem zu erstellenden Benutzer. Die vorherige Implementierung von `AddressPanel` könnte folgendermaßen von `CompoundPropertyModel` profitieren:

```
1 public AddressPanel(String id, Address addr) {
2   super(id, new CompoundPropertyModel<Address>(addr));
3
4   add(new Label("street"));
5   add(new Label("number"));
6   add(new Label("postcode"));
7   add(new Label("city"));
8 }
```

Listing 2.8: Beispiel des `CompoundPropertyModels`

Dieses Mal wird dem gesamten Panel eine Instanz von `CompoundPropertyModel` zugewiesen, welche dabei das `Address`-Objekt erhält. Diejenigen Kindkomponenten ohne eigenes Modell (hier alle) greifen dann auf das Modell der Elternkomponente zu und verwenden dabei ihre ID als Property-Expression.

2.2.4 `LoadableDetachableModel`

Um das `LoadableDetachableModel` zu verstehen, muss man zunächst wissen, dass Wicket am Ende jeder HTTP-Anfrage allen Komponenten und u. a. den dazugehörigen Modellen die Chance gibt, Ressourcen freizugeben. Dies dient dazu die gesamten Komponentenbäume vorheriger Anfragen möglichst platzsparend im Speicher halten zu können bzw. auf die Festplatte schreiben zu können (serialisiert). Auf dieses Konzept wird in Abschnitt 2.7 auf Seite 11 noch genauer eingegangen.

Natürlich ist es nicht immer wünschenswert den gesamten Zustand dabei zu behalten. Erstens müssen alle beteiligten Objekte dafür serialisierbar sein, zweitens kostet es Speicherplatz und drittens ist es möglicherweise erwünscht, dass z. B. stets die aktuellen Daten aus der Datenbank bezogen werden. Aus diesen Gründen gibt es die Klasse `LoadableDetachableModel<T>`. Möchte man sie verwenden, muss lediglich die Methode `protected T load()` implementiert werden, um das Objekt des Modells zu laden. Das `LoadableDetachableModel` sorgt dann dafür, dass diese pro Anfrage höchstens einmal aufgerufen wird und das Objekt am Ende der Anfrage verworfen wird.

2.3 Events

Ähnlich wie in einer Java-Swing Desktopanwendung, reagiert man in einer Wicket-basierten Webanwendung auf Events. Typische Beispiele für Events sind das Anklicken eines Links, Änderung der Auswahl bei einer Combobox oder das Absenden eines Formulars. Im Unterschied zu Swing muss der Entwickler keine Event-Handler selbst anmelden, da dies von Wicket übernommen wird. Stattdessen wird lediglich eine entsprechende Methode implementiert um die Reaktion auf das Event festzulegen:

```java
add(new Link<Void>("logout") {
  @Override
  public void onClick() {
    AuthenticatedWebSession.get().signOut();
    setResponsePage(HomePage.class);
  }
});
```

Listing 2.9: Reaktion auf das onClick-Event eines Links

Wie in Abschnitt 2.7 auf Seite 11 noch genauer geschildert wird, verarbeitet Wicket am Anfang jeder Anfrage Events und rendert erst anschließend die gewünschte Seite. Diese Seite kann durch die Methode `setResponsePage` auch geändert werden.

2.4 Behaviors

Wicket nutzt Behaviors um Komponenten mit Zusatzfunktionalität auszustatten. Dazu bietet die Klasse `Component` eine Methode `add`, mit welcher man beliebig viele Behaviors zu einer Komponente hinzufügen kann. Die Basisklasse für diese ist `Behavior` und stellt diverse Methoden zur Überschreibung bereit. Auf diese Weise kann man beispielsweise auf den Rendervorgang des Heads reagieren, auf den Rendervorgang der Komponente selbst oder auf Ereignisse für die Komponente.

Eine bereits von Wicket bereitgestellte Behavior ist `AttributeAppender` bzw. `AttributeModifier`. Dadurch lassen sich HTML-Attribute dynamisch hinzufügen oder verändern. Möchte man z. B. vor das Löschen von Daten durch einen Link ein Javascript-Popup schalten, lässt sich das folgendermaßen realisieren:

```java
Link<Void> link = new Link<Void>("delete") { /* ... */ };
link.add(new AttributeAppender("onclick", Model.of("return confirm('Sind Sie sicher?')")));
```

Listing 2.10: Ein `AttributeAppender` für ein Javascript-Popup

Die Vorgehensweise es nicht in der Markupdatei unterzubringen ist u. a. dann notwendig, wenn man diese Bestätigung nur unter bestimmten Bedingungen hinzufügen möchte (z. B. wenn es der Benutzer abschalten kann).

Eine weitere typische Verwendung von Behaviors ist es, AJAX-Funktionalität zu Komponenten hinzuzufügen. So lässt sich eine beliebige Komponente in einem bestimmten Intervall via AJAX aktualisieren:

```java
comp.add(new AjaxSelfUpdatingTimerBehavior(Duration.seconds(1)));
```

Listing 2.11: Aktualisierung von Komponenten durch AJAX

2.5 Formulare

Da Formulare ein Kernbestandteil aller ernsthaften Webanwendungen sind, soll deren Handhabung in Wicket-basierten Anwendungen hier kurz erläutert werden. Zu diesem Zweck existiert die Klasse `Form` und Wicket stellt diverse Formular-Komponenten wie `TextField`, `Button`, `Checkbox` und `Select` bereit. Typischerweise wird die `onSubmit`-Methode von `Form` überschrieben um auf die Absendung des Formulars zu reagieren. Auf die bei Formularen stattfindende Konvertierung und Validierung von Daten soll in diesem Abschnitt etwas genauer eingegangen werden.

2.5.1 Konvertierung

Üblicherweise stehen ein oder mehrere Domänenobjekte hinter einem Formular, auf deren Attribute über die Formular-Komponenten (via Modell) zugegriffen wird. Natürlich sollen die Attribute der Domänenobjekte den jeweils passenden Datentyp besitzen. Bei der Befüllung der HTML-Formularelemente während des Rendervorgangs werden diese jedoch als String benötigt und auch nach Absendung des Formulars kommen diese per HTTP in Form eines Strings am Server an. Zu diesem Zweck verwendet Wicket Konverter, deren Grundlagen das Interface `IConverter` ist:

```
1 public interface IConverter<C> extends IClusterable {
2   C convertToObject(String value, Locale locale);
3   String convertToString(C value, Locale locale);
4 }
```

Listing 2.12: Das Interface `IConverter`

Konverter können entweder global für die gesamte Anwendung oder für einzelne Komponenten festgelegt werden. Diese werden jeweils für ein bestimmtes `Class`-Objekt registriert. Bei jeder Formular-Komponente kann über `setType(Class<?> type)` spezifiziert werden in welchen Typ der String konvertiert werden soll. Dies geschieht typischerweise automatisch über Reflection durch Verwendung von `PropertyModel` oder dadurch, dass die Formular-Komponente dies selbst festlegt (z. B. wird der Zustand einer `Checkbox` stets durch ein `Boolean` repräsentiert), so dass `setType` nur in Ausnahmefällen benötigt wird.

2.5.2 Validierung

Häufig unterliegen die Daten, die per Formular erfasst werden sollen, technischen oder domänenspezifischen Einschränkungen (eine E-Mail Adresse muss gültig sein, ein Benutzername muss mindestens 5 Zeichen lang sein, ...). Diese Anforderung regelt Wicket über Validatoren, die den Formular-Komponenten angehängt werden können. Alle Validatoren, die zu dem abgesendeten Formular gehören, werden vor Aufruf der `onSubmit`-Methode des Formulars ausgeführt. Nur wenn dabei kein Fehler auftritt, wird `onSubmit` aufgerufen. Fehler, Warnungen, Hinweise und anderes werden in der Session gespeichert und können mit einem `FeedbackPanel` bei Bedarf angezeigt werden.

2.5.3 Beispiel

Um die vorherigen Ausführungen zu konkretisieren, soll folgendes Beispiel betrachtet werden. Wie in einigen anderen bisher verwendeten Beispielen, wird hier ebenfalls die Klasse `Address` eingesetzt.

Dieses mal soll sie bearbeitet werden können:

```java
public class EditAddressPanel extends Panel {
  public EditAddressPanel(String id, Address addr) {
    super(id);

    Form<Address> form = new Form<Address>("form",
        new CompoundPropertyModel<Address>(addr)) {
      @Override
      protected void onSubmit() {
        // Adresse speichern
      }
    };

    form.add(new RequiredTextField<String>("street"));
    form.add(
      new RequiredTextField<String>("number")
        .add(new PatternValidator("^\\d+[a-z]?$"))
    );
    form.add(new RequiredTextField<Integer>("zipcode"));
    form.add(new RequiredTextField<String>("city"));
    add(form);
  }
}
```

Listing 2.13: Ein Panel mit Formular zur Bearbeitung einer Adresse

Wie das Listing zeigt, wird dem Formular ein `CompoundPropertyModel` mit der Adresse zugewiesen und es werden für die einzelnen Attribute der Adresse Formular-Komponenten hinzugefügt. Da es sich bei allen Datentypen um Java-Standarddatentypen handelt (`String` und `Integer`), ist kein Aufruf von `setType` notwendig um die Konvertierung festzulegen. Die Validierung soll in diesem Fall zum einen alle Felder als Pflichtfelder ansehen und zum anderen nur Hausnummern zulassen, die einem bestimmten Muster entsprechend. Für ersteres kommt die Klasse `RequiredTextField` zum Einsatz und für letzteres wird ein `PatternValidator` verwendet. In der überschriebenen Methode `onSubmit` kann davon ausgegangen werden, dass alle Validatoren keinen Fehler gemeldet haben, so dass direkt die gewünschte Aktion ausgeführt werden kann, d. h. beispielsweise die Persistierung des (bereits vollständig gefüllten) Objekts `addr`.

2.6 Ressourcen

Wicket bietet umfangreiche Möglichkeiten Ressourcen zu verwenden, auf deren Grundlagen in diesem Abschnitt eingegangen werden soll. Unter Ressourcen werden in diesem Fall Markupdateien, Bilder, CSS-Dateien, Javascript-Dateien etc. verstanden. Markupdateien werden im Gegensatz zu den anderen Ressourcen nicht explizit angegeben, sondern bestimmte Komponenten wie z. B. `Panel` und `WebPage` erwarten (standardmäßig) eine dazugehörige und entsprechend benannte Markupdatei im gleichen Verzeichnis. Die übrigen genannten Ressourcen können auf verschiedene Art und Weise referenziert werden. Es kann z. B. die Klasse `PackageResource` verwendet werden um eine Ressource anzugeben, die innerhalb des gleichen Verzeichnisses wie eine spezifizierte Klasse liegt. Sie können aber auch direkt im Markup angegeben und mit `<wicket:link>` umgeben werden, so dass Wicket sie im Verzeichnis der dazugehörigen Komponente erwartet und den Pfad zur Ressource entsprechend anpasst. Ohne Verwendung von `<wicket:link>` hat Wicket keine Kenntnis über die Ressource und dementsprechend muss sie an der angegebenen Stelle im Verzeichnis der Webanwendung liegen.

Abgesehen von dem letzten Fall, kann jede Ressource in unterschiedlichen Varianten zur Verfügung

stehen, abhängig von *Style* und *Locale*. Wicket realisiert dies, indem es auf bestimmte Art und Weise verschiedene Dateinamensvarianten durchprobiert bis eine existierende Datei gefunden wurde und setzt somit auf das Paradigma „Convention over configuration“. Der Algorithmus probiert folgende Varianten [2]:

```
<name>_<style>_<locale>.<extension>
<name>_<locale>.<extension>
<name>_<style>.<extension>
<name>.<extension>
```

Beispielsweise könnte es für die bereits erwähnte Komponente `AddressPanel` die Markupdateien `AddressPanel_christmas_de_DE.html` und `AddressPanel.html` geben. Erstere würde verwendet werden, wenn der Style „christmas“ aktiviert ist und das Locale „de_DE“ ist, und letztere in allen anderen Fällen.

2.7 Anfrage-Bearbeitung

Am Ende der Beschreibung der Architektur von Wicket soll noch einmal ein Überblick über die Bearbeitung von Anfragen gegeben werden. Wicket erledigt dies in mehreren Schritten [6, S. 204]:

1. Dekodierung der URL,
2. Verarbeitung von Events,
3. Generierung der Antwort und
4. Freigabe von Ressourcen (Detach Phase).

Wie in der Einleitung erwähnt, hat Wicket das Ziel den Anwendungsentwickler von HTTP so weit wie möglich abzuschirmen. Ein Aspekt dabei ist, dass die Enkodierung und Dekodierung von URLs vollständig durch das Framework übernommen wird. D. h. der Anwendungsentwickler linkt zu bestimmten `WebPage`-Instanzen, gibt ihnen ggf. Parameter mit, oder arbeitet mit Events, muss sich jedoch nicht darum kümmern wie genau eine `WebPage` aufgerufen wird, die Parameter übergeben werden oder die Events ausgelöst werden.

Wicket unterscheidet dabei zwischen *Bookmarkable Pages* und *zustandsbehafteten Pages*. Erstere sind unabhängig von der Session, d.h. von dem serverseitigen Zustand, und können somit als Bookmark gespeichert werden. Letztere hängen von der Session ab und können daher nicht zu beliebigen Zeitpunkten aufgerufen werden. [6, S. 205,206] Ereignisse werden immer durch zustandsbehaftete Anfragen ausgelöst, wohingegen für Links zu Seiten der Anwendung sowohl Bookmarkable Pages als auch zustandsbehaftete Pages zum Einsatz kommen können. Bei diesen liegt es also in der Hand des Anwendungsentwicklers die passende Variante zu wählen. Beispielsweise könnte folgendermaßen auf die Login-Seite verwiesen werden:

```java
add(new BookmarkablePageLink<example.LoginPage>("login", example.LoginPage.class));
```

```html
<a wicket:id="login">Login</a>
```

Die Klasse `BookmarkablePageLink` würde dann dem dazugehörigem Link im Markup eine URL wie `/bookmarkable/example.LoginPage` geben. Auf diese Weise kann Wicket dann im ersten Schritt der Anfrage-Bearbeitung sehen, dass die Page `example.LoginPage` gerendert werden soll.

Im zweiten Schritt werden ggf. Events verarbeitet – jedoch nur bei zustandsbehafteten Pages. Ein Formular könnte z. B. die URL `page?8-2.IFormSubmitListener-registerForm` als Ziel besitzen. Anhand dessen sieht Wicket das auszulösende Ereignis in Form der dazugehörigen Schnittstelle (`IFormSubmitListener`) und die ID[3] der Komponente (`registerForm`). Daraufhin würde also die entsprechende Methode (hier `onSubmit`) dieser Komponente aufgerufen werden. In dieser kann ggf. die zu rendernde Page via `setResponsePage` geändert werden.

Anschließend wird die Antwort für die Anfrage generiert. Typischerweise wird HTML produziert, indem die Methode `renderPage` der zu rendernden Page aufgerufen wird, welche dann dafür sorgt, dass der gesamte Komponentenbaum rekursiv den entsprechenden HTML-Code generiert. Alternativ kann auch eine Weiterleitung veranlasst werden oder eine AJAX-Anfrage mit XML-Code beantwortet werden.

Der letzte Schritt ist die sogenannte *Detach Phase*. Wie bereits in Abschnitt 2.2.4 auf Seite 7 erwähnt, dient diese dazu Ressourcen freizugeben. Wicket speichert den gesamten Komponentenbaum jeder Anfrage (bis zu einem gewissen Limit) in einer sogenannten *Page Map* und vergibt dabei eine ID um später darauf zurückgreifen zu können. Zudem veranlasst Wicket den Browser bei jeder Anfrage eine ID per HTTP-GET mitzuschicken. Auf diese Weise hat jede Anfrage eine andere URL, so dass im Browser die Buttons „Zurück“ und „Vorwärts“ verwendet werden können. Denn Wicket kann anhand der ID einfach den dazugehörige Komponentenbaum aus der Page Map heraussuchen und diesen rendern, so dass stets genau mit dem Zustand der damaligen Anfrage gearbeitet wird. Um den Ressourcenverbrauch durch die Speicherung der Komponentenbäume gering zu halten, können in der Detach Phase nicht mehr benötigte Ressourcen freigegeben werden. Dazu wird die Methode `detach` von `WebPage` aufgerufen, wodurch bei allen Komponenten im Baum und den dazugehörigen Modellen ebenfalls diese Methode aufgerufen wird.

[3] Genauer gesagt ist es der Pfad zur Komponente, bestehend aus den IDs der Komponenten im Baum zu derjenigen, die das Event empfangen soll.

3 Analyse der Architektur

Nachdem nun die wichtigsten Aspekte der Architektur von Apache Wicket beschrieben wurden, soll dieser Abschnitt versuchen sie zu analysieren und ihre Konsequenzen zu ermitteln.

3.1 Komponenten

Das interessanteste und wichtigste an Wicket sind sicherlich die Komponenten und ihre Beziehung zum Markup. Bei erster Betrachtung entsteht möglicherweise der Eindruck, dass die Anwendungslogik und die Benutzeroberfäche in Wicket miteinander vermischt werden. Denn obwohl die Markupdateien das Design festlegen, sind sie stark an die dazugehörigen Komponenten, also den Java-Code, gekoppelt. Immerhin muss der Komponenten-Baum im Java-Code stets mit dem Baum im Markup übereinstimmen. Zudem werden einige Aspekte der Benutzeroberfläche im Markup festgelegt, andere hingegen im Java-Code.

Obwohl dieser Eindruck nicht ganz falsch ist, ist der Blickwinkel darauf ein anderer. Die Idee ist das Design von der restlichen Anwendung zu trennen, weil typischerweise Designer und Entwickler gemeinsam an einer Webanwendung arbeiten. Erstere sind dabei für HTML, CSS, Bilder, usw. zuständig, nicht aber für die UI-Logik. Diese wird von den Entwicklern festgelegt. Aus diesem Grund trennt Wicket das Markup von der restlichen Anwendung und befreit den Designer zudem von der Auseinandersetzung mit Template-Sprachen, Expression-Languages, Tag-Bibliotheken o. ä. . Dies hat außerdem den Vorteil, dass die Entwickler ausschließlich mit Java-Code arbeiten und von dessen Eigenschaften bei der Implementierung der UI-Logik profitieren (Ausdrucksstärke, Typsicherheit, . . .). Durch diese Trennung kann sich also jeder auf seine Aufgabe konzentrieren und muss nicht diverse Sprachen beherrschen. Dem Entwickler ist es dabei freigestellt ob er in die Komponenten auch Geschäftslogik, Persistenzlogik, usw. integriert oder ob diese nur die UI-Logik festlegen und dabei lediglich eine separate Geschäftslogik- oder Persistenzschicht verwenden.

Auf der anderen Seite führt es dazu, dass bei Änderungen in der Regel sowohl das Markup als auch der Java-Code angefasst werden muss. Zum einen, weil im Markup durch die erzwungene Übereinstimmung der Bäume die Flexibilität schwindet (es ist z. B. nicht möglich eine Komponente zu einer anderen Elternkomponente hinzuzufügen). Zum anderen, weil keine Template-Sprache o. ä. vorhanden ist, so dass z. B. eine nachträgliche Anforderung wie einen bestimmten Bereich einer Seite nur für angemeldete Benutzer sichtbar zu machen, nicht ohne Änderung des Java-Codes realisiert werden kann.

Wicket liefert für die verschiedenen HTML-Elemente bereits vorgefertigte Komponenten mit, welche in fast allen Fällen sehr intuitiv verwendbar sind. Beispielsweise wird ein `Link` einem `a`-Element zugeordnet, ein `Image` einem `img`-Element oder ein `TextField` einem `input`-Element mit dem Attribut `type="text"`. Aber standardmäßig wird diese Zuordnung nicht zur Compilezeit geprüft[1],

[1] Auch wenn ein Tool dafür vorstellbar wäre.

so dass derartige Fehler erst zur Laufzeit erkannt werden. Analog dazu können Elemente im Markup oder Komponenten im Java-Code vergessen werden. Diese Fehler können jedoch in der Regel durch die guten Fehlermeldungen von Wicket schnell behoben werden. Es gibt allerdings auch Fälle, indem nicht unbedingt klar ist, an welches HTML-Element eine Komponente gebunden werden soll. Soll beispielsweise `DataTable` einem `table`-Element oder einem `div`-Element zugeordnet werden? Beides ist theoretisch denkbar, d. h. es ist sowohl möglich, dass `DataTable` nur den Inhalt des `table`-Elements liefert als auch, dass es die gesamte Tabelle mit ggf. Elementen drumherum bereitstellt. Im Zweifelsfall muss ein Blick in die Dokumentation der Komponente oder sogar in ihren Markupcode geworfen werden.

Die Verwendung von fertigen Komponenten, die in diesem Fall als Blackboxes betrachtet werden, hat natürlich zur Folge, dass der Entwickler zu einem gewissen Grad die Kontrolle über den produzierten HTML-, CSS- und Javascript-Code und auch über Bilder und andere Ressourcen verliert. So ist in einigen Fällen ggf. ein Blick in den im Browser angekommenen Code oder sogar den Code der Komponente selbst angebracht um zu verhindern, dass eine Webseite mit Ressourcen überladen wird. Der Kontrollverlust über den HTML-Code fällt bei Wicket allerdings geringer aus als bei Tag-Bibliothek-basierten Frameworks, da HTML-Elemente von Komponente sozusagen nur angereichert werden. So kann in den meisten Fällen auf beliebige Attribute zurückgegriffen werden. Z. B. kann ein a-Element ohne Wissen von `Link` das Attribut `target="_blank"` erhalten. Dies funktioniert natürlich nicht, wenn eine Komponente ein oder mehrere gesamte HTML-Elemente selbst liefert, wie z. B. bei dem `AddressPanel`. In diesem Fall bietet Wicket jedoch die Möglichkeit von der Komponentenklasse abzuleiten und eine eigene Markupdatei dafür zur Verfügung zu stellen, die das gesamte Markup der Elternklasse mit den gewünschten Änderungen enthält.

Auf der anderen Seite gestaltet sich die Entwicklung von eigenen Komponenten (in der Regel Panels, die mehrere häufig in dieser Konstellation benötigte Komponenten zusammenfassen) derart einfach, dass oft davon Gebrauch gemacht wird. Dies dient zum einen zur Verbesserung der Kohäsion, wenn ein Teil einer Seite eine logische Einheit bildet und somit in ein eigenes Panel verlagert wird. Zum anderen wird dadurch Code-Duplizierung verhindert, wenn dieser Teil mehrfach auftritt. Die Wiederverwendung führt indirekt auch zur Einheitlichkeit der Benutzeroberfläche. Eine weitere Konsequenz der einfachen Erstellung von eigenen Komponenten ist, dass dies ohne Probleme im Nachhinein durchgeführt werden kann, wenn sich später herausstellt, dass ein Teil einer Seite extrahiert werden soll. Wegen des geringen Änderungsaufwands müssen dabei lediglich softwaretechnische Gründe abgewogen werden und nicht der Zeitaufwand (vergleiche Listing 2.1 und 2.2 mit Listing 2.4 und 2.5).

3.2 Modelle

Die Modelle in Wicket entsprechen weder den Domänenobjekten noch enthalten sie Geschäftslogik. Anstelle dessen können sie als Vermittler zwischen Komponenten und Domänenobjekten betrachtet werden. Durch sie werden also bestehende Daten an die Komponenten gebunden und können – sofern erwünscht – auch von ihnen verändert werden. Die Komponenten wissen dabei nicht woher die Daten kommen oder wohin Änderungen geschrieben werden und benötigen dieses Wissen auch nicht. Aus diesem Grund ist Wicket unabhängig von der verwendeten Persistenztechnologie.

Eine weitere Konsequenz des Modellkonzeptes ist, dass man in Wicket-basierten Anwendungen typischerweise keine eigenen (nicht-anonymen) Modell-Klassen implementiert – z. B. eine für jedes

Domänenobjekt oder eine für jeden Anwendungsfall. Anstelle dessen bedient man sich bei bereits bestehenden Modell-Implementierungen wie `PropertyModel`, `CompoundPropertyModel` und `LoadableDetachableModel`, erstellt lediglich ein Objekt dieser Klasse und überreicht es der Komponente. Letztere Modellvariante verlangt zwar eine Implementierung der Methode `load`, aber diese gestaltet sich in der Regel sehr kurz, so dass oftmals eine anonyme Klasse verwendet wird. Das heißt auch, dass Modelle typischerweise nicht wiederverwendet werden, da sie nur zur Bindung von Daten an Komponenten dienen und keine Logik enthalten, die es wert wäre wiederverwendet zu werden.

Gerade die auf Property-Expressions basierenden Modelle wie beispielsweise `PropertyModel` und `CompoundPropertyModel` ersparen dem Entwickler viel Arbeit. In beiden Fällen aus dem Grund, dass nicht für jede Eigenschaft eines Domänenobjektes, die an eine Komponente gebunden werden soll, eine eigene Modell-Implementierung benötigt wird. Das `CompoundPropertyModel` bietet zum einen den zusätzlichen Vorteil, dass Tipparbeit durch das auf „Convention over configuration“ basierende Konzept, die Komponenten-ID als Property-Expression heranzuziehen, entfällt. Immerhin sind diese typischerweise ohnehin identisch. Zum anderen, weil das Binden eines ganzen Domänenobjektes an eine Komponente sehr intuitiv ist: Ein Bestellformular bekommt ein Bestellungsobjekt, da es die Bestellung repräsentiert.

Ein Nachteil der Property-Expression ist die fehlende Validierung durch den Compiler. Für den Compiler ist es nur ein String mit beliebigem Inhalt, so dass erst zur Laufzeit erkannt wird, ob eine Property-Expression gültig ist oder nicht. Dies ist vor allem ein Problem bei nachträglichen Änderungen an Domänenobjekten wie der Umbenennung eines Attributs. Aus dem Grund ist es empfehlenswert die Domänenobjekte sorgfältig zu planen, so dass im Nachhinein nur wenig Änderungen erforderlich sind. Abhilfe kann hier auch der intensive Einsatz von Unit-Tests schaffen, durch diese derartige Fehler ohne manuelle Prüfung des Entwicklers erkannt werden [7, S. 74].

3.3 Events

In Wicket gibt es keinen Controller, der festlegt auf welche Anfrage wie reagiert wird. Dies ist nicht erforderlich, weil der Event-Mechanismus des Frameworks diese Aufgabe übernimmt. D. h. das Framework sorgt dafür, dass abhängig von der Anfrage-URL ein bestimmtes Ereignis bei einer bestimmten Komponente ausgelöst wird. Dadurch ist keine Konfiguration in Form von XML oder Java-Code nötig. Auf der anderen Seite hat man dadurch nicht die gesamte Navigation auf einen Blick, wie bei einer zentralen XML-Datei, sondern sie ist über die gesamte Anwendung verstreut und sozusagen fest im Java-Code bzw. im Markup verdrahtet. Dies bedeutet aber auch, dass der Entwickler nicht die Syntax und Semantik von speziellen XML-Dateien für diesen Zweck lernen muss und das er von der statischen Analysierbarkeit von Java profitiert, wenn er die Navigation dort festlegt.

Die Vorgehensweise auf Events zu reagieren ist intuitiv und eine gute Abstraktion, die die Details des HTTP-Protokolls und die Kodierung der URLs vor dem Anwendungsentwickler verbirgt. Die Ähnlichkeit zu der Entwicklung von Desktop-Anwendungen mit Java-Swing hat zudem den Vorteil, dass viele Entwickler damit bereits vertraut sind.

Auf der anderen Seite lässt sich darüber streiten, ob die Art und Weise wie auf Events reagiert wird nicht den Entwickler dazu verleitet komplexe Geschäftslogik an dieser Stelle zu implementieren. Denn die Event-Methoden wie `onClick` oder `onSubmit` sind in der Komponentenklasse abstrakt und

müssen somit von dem Anwendungsentwickler noch implementiert werden. Da eine Wiederverwendung häufig nicht möglich oder nicht lohnenswert ist, wird dies typischerweise durch eine anonyme Klasse realisiert. Es wird also die Reaktion auf das Event direkt in die UI-Logik integriert. Es kann jedoch auch argumentiert werden, dass sie genau dort hingehört, denn auf angemessenem Abstraktionslevel ist die Reaktion teil der UI-Logik. Beispielsweise kann das Ausloggen eines Benutzers und Weiterleitung auf die Startseite durchaus als UI-Logik bezeichnet werden. Nur sollte der eigentliche Auslogvorgang besser separat davon implementiert werden und in der Reaktion auf das Event lediglich angestoßen werden.

3.4 Behaviors

Durch Behaviors lassen sich Komponenten mit Zusatzfunktionalität ausstatten. Sie lassen sich sehr gut wiederverwenden, da sie von der Komponente, der sie hinzugefügt werden, keine Kenntnis haben und somit unabhängig von ihr sind bzw. sein können.

Ihre Verwendung birgt jedoch die Gefahr Logik und Design zu vermischen, da sie im Java-Code festgelegt werden. Beispielsweise ist es nicht empfehlenswert das Attribut `style` zu verändern. Stattdessen sollte in diesem Fall das Attribut `class` gesetzt werden und das Design per CSS spezifiziert werden.

Ein Nachteil des Behavior-Konzeptes ist, dass sich Behaviors gegenseitig stören können. Denn es können beliebig viele Behaviors zu einer Komponente hinzugefügt werden, wobei die einzelnen keine Kenntnis voneinander besitzen. Beispielsweise ist das Ergebnis bei mehrfacher Hinzufügung von `AttributeModifiern` von deren Reihenfolge abhängig. D. h. der letzte `AttributeModifier` könnte die zuvor vorgenommenen Änderungen wieder rückgängig machen, indem er das Attribut überschreibt. Dies kann vor allem bei Third Party Komponenten auftreten, die bereits eigene Behaviors besitzen, von denen der Verwender jedoch möglicherweise nichts weiß.

3.5 Ressourcen

Ähnlich wie bei den Komponenten wirkt es möglicherweise zunächst so, als ob Design und Logik vermischt werden, weil Java-Code, Markupdateien und oftmals sogar Bilder und andere Ressourcen in ein und demselben Verzeichnis liegen. Immerhin ist es in vielen anderen Frameworks üblich die Markupdateien und andere Ressourcen für die Benutzeroberfläche in einem anderen Verzeichnis als die Anwendungslogik zu halten.

Dies ist jedoch nicht wirklich der Fall und es gibt einen guten Grund für diese Vorgehensweise. Zunächst handelt es sich bei den Komponentenklassen, wie bereits erwähnt, um UI-Logik, d. h. diese gehören zur Benutzeroberfläche dazu. Daneben ist die Idee hinter diesem Konzept, dass man auf diese Weise Komponenten sehr einfach weitergeben kann. Dadurch dass die Komponenten-Klasse, die Markupdateien, Propertydateien, CSS-Dateien, Bilder und alle weiteren benötigten Ressourcen in einem Java-Paket (ggf. mit Unterverzeichnissen) liegen und relativ zu diesem referenziert werden, können diese einfach in ein JAR-Archiv gepackt werden und weitergegeben werden. Um die Komponente zu verwenden muss lediglich die JAR-Datei eingebunden werden. Es ist also nicht notwendig die Ressourcen in andere Verzeichnisse zu kopieren, die Pfade zu konfigurieren usw..

3.6 Anfrage-Bearbeitung

Das Konzept zur Bearbeitung von Anfragen in Wicket erspart dem Anwendungsentwickler viel Arbeit und schirmt ihn von den Details des HTTP-Protokolls ab. So muss er sich z. B. nicht um die Kodierung von URLs kümmern. Auf der anderen Seite führt die Art und Weise wie Links konstruiert werden und die Pages instanziert werden dazu, dass Einsteiger in die objektorientierte Programmierung möglicherweise verwirrt werden. Denn da der Entwickler nur in Ausnahmefällen eine Instanz einer Page erstellt und wenig davon mitbekommt, dass es auch mehrere Objekte einer Page geben kann, wird der Unterschied zwischen Klassen und Objekten evtl. nicht klar. Der Markup-Vererbungsmechanismus, welcher zwar korrekt im Sinne der Objektorientierung umgesetzt ist, aber relativ kompliziert ist, macht dieses Problem noch größer. Aus diesem Grund sollten Entwickler schon vor der Beschäftigung mit Wicket die Konzepte der Objektorientierung vollständig verstanden haben.

Ein häufiges Problem in Webanwendungen sind die Buttons „Zurück" und „Vorwärts" im Browser [3], durch welche der Benutzer aus dem vom Entwickler vorgesehenen Navigationsablauf ausbrechen kann. So müssen Webanwendungen immer damit rechnen, dass ein Benutzer z.B. während eines komplexen Bestellvorgangs eine Seite zurück navigiert. Wicket löst dieses Problem durch die Page ID und die Page Map und ohne das der Anwendungsentwickler sich darum kümmern muss.

Ein weiterer Vorteil dieses Konzeptes ist die Erhöhung der Sicherheit. Denn um z. B. ein Event zur Löschung von Daten auszulösen, muss sowohl die Session gestohlen werden als auch die richtige Page ID erraten werden. Dadurch gestaltet es sich deutlich schwerer als bei anderen Frameworks. [5, S. 268]

4 Fazit

Zusammenfassend lässt sich sagen, dass sich mit Apache Wicket sehr gut arbeiten lässt. Es besticht dabei vor allem durch seine Einfachheit, konzeptionelle Eleganz und der Abschirmung des Entwicklers vor der Funktionsweise von HTTP. Gerade Swing-Entwicklern dürfte der Einstieg leicht fallen aufgrund vieler Ähnlichkeiten.

Durch die Trennung zwischen dem Design und der restlichen Anwendung eignet es sich sehr gut für die Zusammenarbeit zwischen Entwicklern und Designern. Auf der anderen Seite bedeutet diese Trennung eine Verteilung der Benutzeroberfläche auf Java-Code und Markup. Das führt dazu, dass Wicket vermutlich keine gute Wahl ist, wenn der Kunde selbst Änderungen an der Benutzeroberfläche durchführen möchte, ohne dabei den Java-Code anfassen zu müssen. Bei Verwendung einer Templatesprache hat der Kunde größere Flexibilität bei der Änderung und kann so eben auch (bis zu einem gewissen Grad) die UI-Logik verändern.

Ein weiterer kleiner Kritikpunkt, der bei der Arbeit mit dem Framework sichtbar wurde, ist die bisher eher dürftige Dokumentation zur Umsetzung speziellerer Anforderungen mit Wicket. So existiert zwar eine sehr aktive und hilfsbereite Mailing-Liste[1], aber eine sorgfältig geschriebene Frage-Antwort-Liste zu häufigen Problemen oder ähnlichem sucht man vergeblich. Das bisherige Wiki[2] ist zwar eine gute Basis, aber im derzeitigen Stand leider zu unstrukturiert, oft nicht aktuell und unvollständig.

[1] http://wicket.apache.org/help/email.html
[2] https://cwiki.apache.org/WICKET/index.html

Literaturverzeichnis

[1] *Apache Wicket*. http://de.wikipedia.org/w/index.php?title=Apache_Wicket&oldid=94879907,

[2] *Localization and Skinning of Applications*. https://cwiki.apache.org/WICKET/localization-and-skinning-of-applications.html,

[3] *Spring Web Flows: Back Button Handling*. http://www.ervacon.com/products/springwebflow/backButtonHandling.html,

[4] *Tim Berners-Lee*. http://de.wikipedia.org/w/index.php?title=Tim_Berners-Lee&oldid=93041601,

[5] DASHORST, M. ; HILLENIUS, E.: *Wicket in Action*. Manning, 2008 (Manning Pubs Co Series). – ISBN 9781932394986

[6] FÖRTHER, R. ; MENZEL, C.E. ; SIEFART, O.: *Wicket: Komponentenbasierte Webanwendungen in Java*. Dpunkt-Verl., 2010. – ISBN 9783898645690

[7] MOSMANN, M.: *Praxisbuch Wicket: Professionelle Web-2.0-Anwendungen entwickeln*. Hanser Fachbuchverlag, 2009. – ISBN 9783446419094