Mehr als die Summe seiner Teile

Vaadin + CDI = MVP

Oliver Damm

Vaadin ist eine interessante Alternative zu JavaServer Faces (JSF) im JEE-Stack. Es lassen sich damit Weboberflächen effizient entwickeln und man kann die Vorteile von purer Java-Entwicklung genießen. Trotzdem muss sich der Entwickler Gedanken über die Architektur seiner Webanwendung machen. Wie integriere ich Vaadin in den JEE-Stack? Wie werden Oberflächen voneinander entkoppelt und wie kann ich Oberflächen wiederverwendbar machen? Die Antworten auf diese Fragen wird der Artikel geben, der zunächst auf das Entwurfsmuster Model-View-Presenter (MVP) eingeht und dann eine Umsetzung mit Vaadin und Contexts and Dependency Injection (CDI) vorstellt.

Modell und Ansicht über einen Präsentator entkoppeln

MVP ist ein bewährtes Entwurfsmuster zur Darstellung und Steuerung von Daten auf einer Benutzeroberfläche. Dabei werden Daten, Darstellung und Steuerung in die drei Komponenten

- Model bzw. Modell,
- View bzw. Ansicht und
- Presenter bzw. Präsentator

getrennt. Martin Fowler beschreibt zwei Varianten des Entwurfsmusters: *Supervising Controller* [SuCo] und *Passive View* [PaVi]. Der wesentliche Unterschied zwischen diesen Varianten ist die Datensynchronisierung. Bei Supervising Controller übernimmt die Ansicht die einfache Datensynchronisierung selbst. Komplexere Aktualisierungen übernimmt hingegen der Präsentator. Diese Variante lässt sich gut durch UI-Frameworks implementieren, die Data-Binding [DaBi] unterstützen.

Bei Passive View hingegen übernimmt der Präsentator sämtliche Synchronisierung zwischen Ansicht und Modell. Die Ansicht enthält keinerlei komplexe Logik. Es besteht keine Verbindung zwischen Ansicht und Modell. Abbildung 1 stellt die beiden Varianten grafisch gegenüber.

Die Variante Passive View hat den Nachteil, dass der Implementierungsaufwand für den Präsentator höher ist, da jegliche Datensynchronisierung dort implementiert werden muss. Der Vorteil dieser Variante ist eine erhöhte Testbarkeit. Diese entsteht, da die Komponenten strenger getrennt sind. Zusätzlich steckt die komplexe Logik im Präsentator. Diese lässt sich gut über Unit-Tests verifizieren. Komplexe Logik in der Ansicht muss über Oberflächentests getestet werden, die allerdings in der Regel unhandlicher sind als Unit-Tests (z. B. höhere Ausführungszeit und geringere Stabilität).

Die erhöhte Testbarkeit der Variante Passive View wiegt aus Sicht des Autors den Nachteil der erhöhten Datensynchonisierung auf. Daher wird in diesem Artikel die Variante Passive View verwendet.

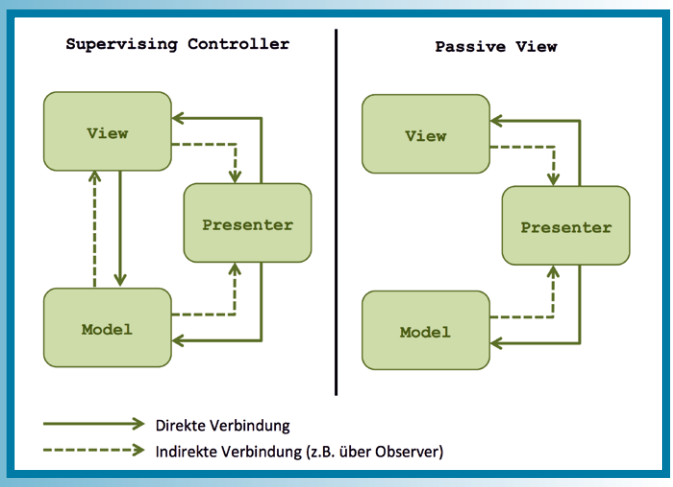

Abb. 1: Supervising Controller und Passive View

Demo-Projekt: Kontaktverwaltung

Die Umsetzung von MVP mit Vaadin und CDI wird anhand einer Demo-Anwendung gezeigt. Der komplette Quellcode [DeAn] steht auf GitHub zur Verfügung. Für das Deployment benötigen Sie einen CDI-fähigen Servlet-Container wie JBoss [JBoss].

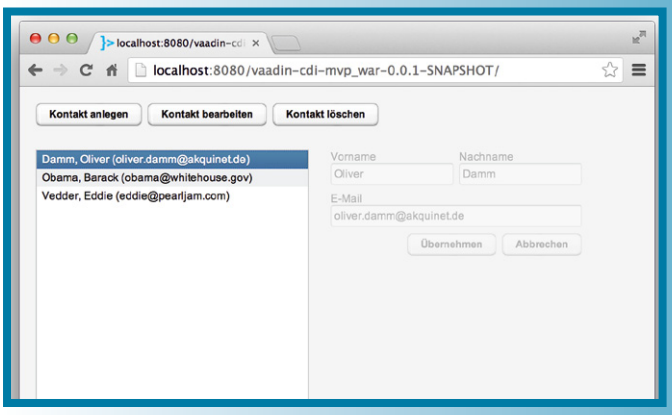

Abb. 2: Demo-Anwendung

Bei der Anwendung handelt sich um eine rudimentäre Kontaktverwaltung. Abbildung 2 zeigt einen Screenshot. Die Oberfläche der Anwendung besteht aus drei Teilen:

- Toolbar mit Buttons zum Anlegen, Editieren und Löschen eines Kontakts,
- Liste aller Kontakte,
- Editor für einen Kontakt.

Die Anwendung ist als Maven-Multimodul-Projekt aufgesetzt. Über die verschiedenen Module werden die drei Teile View, Presenter und Model und deren Abhängigkeiten abgebildet. Wir gehen noch einen Schritt weiter und unterteilen Programmierschnittstelle und Implementierung der jeweiligen Teile in eigene Module. Die resultierenden Module und Abhängigkeiten sind in Abbildung 3 dargestellt.

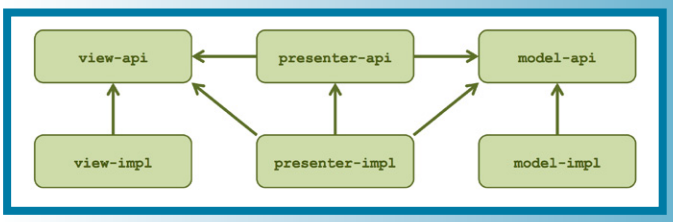

Abb. 3: Maven-Module und -Abhängigkeiten

MVP mit Vaadin und CDI

Nach der anfänglichen Theorie geht es nun in die Praxis, sprich in den Code. Wir beginnen mit der Schnittstelle einer View-Komponente. Diese wird durch das allgemeine Interface `View` (bzw. `ObservableView` zur Unterstützung eines Observers) bereitgestellt und verfügt lediglich über eine Methode `unwrap()`, auf die wir später noch eingehen werden. Für jede konkrete Ansicht der Demo-Anwendung ist ein Interface implementiert, das von `View` erbt und weitere Methoden bereitstellt, um die spezielle Ansicht vom Präsentator aus betreiben zu können. Listing 1 zeigt dies exemplarisch für die Ansicht für die Kontaktliste.

```
public interface ContactListView
    extends ObservableView<ContactListViewObserver> {
  public void setContacts(List<ContactItem> contacts);
  public void select(ContactItem contact);
  public void unselect();
  public ContactItem getSelectedContact();
}
```

Listing 1: Interface ContactListView

Die Implementierung des Interfaces `ContactListView` erfolgt durch die Klasse `ContactListViewComponent`. Ein Ausschnitt dieser Klasse ist in Listing 2 zu sehen. In dieser Klasse begegnen wir das erste Mal CDI-Annotationen. Die Klasse selbst ist mit `@Dependent` annotiert. Dies bedeutet, dass der Lebenszyklus einer CDI-Bean dieser Klasse an den Lebenszyklus des Verwenders gebunden ist. Der Verwender wird in unserem Fall der Präsentator sein.

Die zweite Annotation ist `@PostConstruct` an der Methode `init()`. Diese Methode wird vom CDI-Container aufgerufen, nachdem die Bean vom Container erzeugt und initialisiert wurde.

```
@Dependent
public class ContactListViewComponent extends ListSelect
    implements ContactListView, VaadinView<Component> {
  private static final long serialVersionUID = 1L;
  @PostConstruct
  public void init() {
    setHeight("100%");
    setWidth("300px");
    setNullSelectionAllowed(false);
  }
  ...
}
```

Listing 2: ContactListViewComponent

Pro View-Instanz ist eine Presenter-Instanz vorhanden, die die Ansicht erzeugt und verwaltet. Es besteht also eine 1:1-Beziehung zwischen Ansicht und Präsentator. Wie für die Ansicht steht auch für den Präsentator ein allgemeines Interface `Presenter` zur Verfügung, das von konkreten Präsentatoren implementiert wird. Dieses Interface stellt eine Methode bereit, über die die Ansicht des Präsentators zurückgegeben wird. Listing 3 zeigt dieses Interface.

```
public interface Presenter<V extends View> {
  public V getView();
}
```

Listing 3: Interface Presenter

Wie bei den Ansichten wird für jeden konkreten Präsentator eine Spezialisierung des Interfaces `Presenter` bereitgestellt. Listing 4 zeigt die Implementierung des Interfaces `ContactListPresenterBean` für die Kontaktliste. Interessant ist hier die Annotation `@Inject`. Sie sorgt dafür, dass dem Präsentator von CDI eine Implementierung des Interfaces `ContactListView` zur Verfügung gestellt wird. In der mit `@PostConstruct` annotierten Methode `init()` wird auf die Ansicht zugegriffen. Der CDI-Container hat während der Initialisierung der Bean die Instanzvariable `view` gesetzt, sodass diese nun verwendet werden kann. Die Initialisierung funktioniert in dieser Form nur, wenn es genau eine Implementierung von `ContactListView` gibt. Gibt es mehrere Implementierungen, so müssen noch weitere Informationen bereitgestellt werden. Dazu später mehr, wenn es um die Wiederverwendung von Views/Presentern geht.

```
@SessionScoped
public class ContactListPresenterBean
    implements ContactListPresenter, ContactListViewObserver,
               ContactModelObserver, Serializable {
  private static final long serialVersionUID = 1L;
  @Inject
  private ContactListView view;
  @Inject
  private ContactModel contactModel;
  @PostConstruct
  public void init() {
    this.view.setObserver(this);
    this.contactModel.addObserver(this);
  }
  @Override
  public ContactListView getView() {
    return this.view;
  }
  ...
}
```

Listing 4: ContactListPresenterBean

Die Klasse selbst ist mit `@SessionScoped` annotiert. An allen Stellen, an denen ein `ContactListPresenter` per `@Inject` injiziert wird, erhält man damit die gleiche Instanz, wenn der Code im Kontext der gleichen Session ausgeführt wird. Im Falle einer Vaadin-Anwendung handelt es sich konkret um die gleiche HTTP-Session. CDI-Beans, die SessionScoped sind, müssen serialisierbar sein, damit der Container diese Beans im Falle einer Passivierung der Session wegschreiben kann.

In der Methode `init()` trägt sich der Präsentator als Observer bzw. Beobachter [Obs] bei der Ansicht ein. Dies entspricht der gestrichelten Linie zwischen View und Presenter in Abbildung 1. Benutzeraktionen werden in der Ansicht über Listener an den Vaadin-UI-Komponenten verarbeitet und an den als Observer registrierten Presenter weitergeleitet.

Wie in Abbildung 1 zu sehen ist, existieren zwischen Presenter und Model die gleichen Beziehungen wie zwischen Presenter und View. Allerdings besteht zwischen ihnen keine strenge 1:1-Beziehung. Ein Präsentator kann kein, ein oder mehrere Modelle verwenden. Wie bei den Ansichten können Änderungen im Modell über das Observer-Entwurfsmuster propagiert werden.

Im Beispiel der `ContactListPresenterBean` lassen wir uns ein `ContactModel` injizieren und registrieren den Präsentator als Observer am Modell (Listing 4). In der Demo-Anwendung ist die Implementierung von `ContactModel` durch die Klasse `ContactModelBean` sehr einfach gehalten. Die Kontakte werden in einer Liste im Hauptspeicher gehalten. Damit alle Benutzer die gleiche Sicht auf die Kontakte haben, ist die Klasse `ContactModelBean` mit `@ApplicationScoped` annotiert. Egal, aus welcher HTTP-Session wir uns ein `ContactModel` injizieren lassen, erhalten wir die gleiche Instanz der Implementierung.

Anhand der Liste der Kontakte haben wir nun einen Durchstich von der Ansicht über den Präsentator bis in das Modell gesehen. Nun benötigen wir noch den „Kleber“, um die einzelnen Präsentatoren/Ansichten für Kontaktliste, Kontakteditor und Kontakttoolbar zusammenzuführen. Dafür verwenden wir einen weiteren Präsentator und die dazugehörige Ansicht.

Listing 5 zeigt die interessanten Stellen der Implementierung der Klasse `ContactsViewComponent`. Aufgabe der Klasse ist

es, die Vaadin-UI-Komponenten der drei Ansichten für Toolbar, Liste und Editor anzuordnen. Dafür werden zunächst in der Methode `init()` drei Platzhalter erzeugt und mit Hilfe von Vaadin-Layouts angeordnet. Die Klasse `ContactsViewComponent` implementiert drei Methoden (u. a. `setToolbarView()`) des Interfaces `ContactsView`, mit deren Hilfe die tatsächlichen Komponenten gesetzt werden können.

Nun kommt die zu Beginn des Artikels erwähnte Methode `unwrap()` des Interfaces `View` ins Spiel. Für das Setzen der Unteransichten ist der Präsentator zuständig. Dieser kennt allerdings nur das Interface `View`, aber keine Implementierungsdetails, insbesondere keine Vaadin-Klassen. Diese werden aber benötigt, um zum Beispiel die Vaadin-Komponente der Ansicht in ein Layout einzufügen. Wie in der Methode `setToolbarView()` zu sehen ist, wird die Methode `unwrap()` verwendet, um die Ansicht auf `VaadinView` zu casten. Dieses Interface wird von allen Implementierungen einer Ansicht zusätzlich implementiert und verfügt über die Methode `getComponent()`, die das zentrale Vaadin-Interface `Component` als Rückgabetyp hat. Dadurch erhalten die Ansichten Zugriff auf die Vaadin-UI-Komponente anderer Ansichten und können diese an der entsprechenden Stelle des Layouts setzen.

Die Implementierung `ContactsPresenterBean` des dazugehörigen Präsentators gestaltet sich recht einfach: Es werden die drei Präsentatoren für Toolbar, Liste und Editor per `@Inject` injiziert. In der Methode `init()` werden dann in der Ansicht des Präsentators die drei Ansichten der Unterpräsentatoren gesetzt.

Die Klassen `ContactPresenterBean` und `ContactViewComponent` stellen also ein Bindeglied dar und verwenden die Schnittstellen der drei Unteransichten bzw. Unterpräsentatoren. Diese kennen sich untereinander nicht. Um die Wiederverwendbarkeit zu erhöhen, sollte dies auch so bleiben. Wie dennoch eine Interaktion zwischen Präsentatoren erreicht werden kann, wird im nächsten Abschnitt vorgestellt.

```
@Dependent
public class ContactsViewComponent extends VerticalLayout
    implements ContactsView, VaadinView<Component> {
  private static final longserialVersionUID = 1L;
  private Component componentToolbar;
  private Component componentList;
  private Component componentEditor;
  private HorizontalLayout layoutListEditor;

  @PostConstruct
  public void init() {
    this.componentToolbar = new Label();
    this.componentList = new Label();
    this.componentEditor = new Label();

    // Layout
    ...
  }
  @Override
  public<V> V unwrap(Class<V> type) {
    return type.cast(this);
  }
  @Override
  public void setToolbarView(ContactToolbarView view) {
    Component component = view.unwrap(VaadinView.class).getComponent();
    replaceComponent(this.componentToolbar, component);
    this.componentToolbar = component;
  }
  ...
  @Override
  public Component getComponent() {
    return this;
  }
}}
```

Listing 5: Klasse ContactsViewComponent

Entkopplung von Präsentatoren über Events

Einen Teil der Entkopplung zwischen Präsentatoren können wir über die Modell-Schicht erreichen. Ein Beispiel: Der Präsentator für den Kontakteditor erzeugt bzw. aktualisiert einen Kontakt über das Modell. Das Modell propagiert die Änderung an alle registrierten Observer. Der Präsentator für die Kontaktliste reagiert entsprechend auf die Änderung und aktualisiert dessen Ansicht mit der neuen Kontaktliste.

Spielt die Modell-Schicht in einem Zusammenspiel zwischen zwei Präsentatoren keine Rolle, müssen wir einen anderen Weg finden. An dieser Stelle kommen uns CDI-Events zur Hilfe, die eine Kommunikation zwischen zwei Komponenten erlauben, ohne dass die Komponenten sich gegenseitig kennen müssen.

Ein Beispiel dafür ist die Selektion eines Kontakts in der Kontaktliste. Bei der Selektion sollen im Editor die Details des selektierten Kontakts dargestellt werden. In Listing 6 ist zu sehen, wie in der Klasse `ContactListPresenterBean` ein CDI-Event ausgelöst wird. Zunächst muss die Klasse `javax.enterprise.event.Event` per `@Inject` injiziert werden. Die Klasse hat einen Typ-Parameter. Hier ist der Typ des eigentlichen Events zu verwenden. Ein Event kann eine beliebige Klasse sein. Im Beispiel enthält das Event lediglich die Id des Kontakts, der selektiert wurde, bzw. `null`, wenn ein Kontakt deselektiert wurde. Die Klasse `Event` stellt die Methode `fire()` zur Verfügung, über die das Event ausgelöst wird.

```
@SessionScoped
public class ContactListPresenterBean
    implements ContactListPresenter, ContactListViewObserver,
      ContactModelObserver, Serializable {
  ...
  @Inject
  private Event<ContactSelectedEvent> contactSelectedEvent;
  ...
  @Override
  public void contactListSelected(Long id) {
    this.contactSelectedEvent.fire(
      new ContactSelectedEvent(id));
  }
  @Override
  public void contactListUnselected() {
    this.contactSelectedEvent.fire(
      new ContactSelectedEvent(null));
  }
  ...
}
```

Listing 6: CDI-Events auslösen

Listing 7 zeigt, wie ein Event empfangen wird. Der Empfänger des Events implementiert lediglich eine Methode mit frei wählbaren Namen, die einen einzigen Parameter vom Typ des Events hat. Der Parameter muss mit `@Observes` annotiert werden. Wird bei der Annotation nichts weiter angegeben, wird das ausgelöste Event synchron in der gleichen Transaktion verarbeitet, in der das Event ausgelöst wurde. Es ist wichtig, in welchem Scope sich der Empfänger des Events befindet. Ist der Empfänger `@Dependent`, so wird für jedes ausgelöste Event eine neue Instanz des Empfängers erzeugt. In unserem Fall ist der Empfänger `@SessionScoped`. Es empfängt also die Instanz von `ContactsPresenterBean` das Event, das sich in der gleichen HTTP-Session wie der Auslöser befindet.

```
@SessionScoped
public class ContactsPresenterBean
    implements ContactsPresenter, Serializable {
...
  public void contactSelected(@ObservesContactSelectedEvent event) {
    if (event.getId() != null) {
      this.contactToolbarPresenter.enableButtons(true, true, true);
      final Contact contact = this.contactModel.getContact(
```

```
          event.getId());
        this.contactEditorPresenter.setContact(contact);
        this.contactEditorPresenter.setStates(true, false);
      }
      else {
        this.contactToolbarPresenter.enableButtons(true, false, false);
        this.contactEditorPresenter.setContact(null);
        this.contactEditorPresenter.setStates(false, false);
      }
    }
  }
```

Listing 7: CDI-Events empfangen

Wiederverwendung von Ansicht/Präsentator

Da wir nun wissen, wie Präsentatoren entkoppelt werden, um die Wiederverwendbarkeit zu erhöhen, gehen wir nun kurz darauf ein, wie ein Präsentator mehrfach in einer Session verwendet werden kann. Um das Ziel zu erreichen, wird ein `@SessionScoped` Präsentator mit einem zusätzlichen Qualifier versehen. Ein Qualifier ist eine Annotation, die wiederum mit `@Qualifier` annotiert wird. Er wird sowohl an der konkreten Implementierung eines Präsentators als auch an der Stelle, an der der Präsentator injiziert werden soll, zusätzlich zu `@SessionScoped` bzw. `@Inject` angegeben. Dies funktioniert auch für Events, die ein Präsentator gegebenenfalls auslöst. Events können sowohl beim Injizieren als auch beim Empfang mit einem Qualifier versehen werden. Ein Beispiel dafür ist in den Klassen `AbstractContactToolbarPresenterBean`, `ContactToolbarPresenterBean`, `AlternativeContactToolbarPresenterBean` und `ContactsPresenterBean` der Demo-Anwendung zu finden.

Testen eines Präsentators

Wie zu Beginn des Artikels beschrieben, wird durch die Variante Passive View die Testbarkeit der Präsentatoren erhöht. Am Beispiel der Klasse `ContactEditPresenterBean` werden wird nun sehen, wie ein Test für einen Präsentator implementiert werden kann. Für die Implementierung werden die Frameworks JUnit [Jun], Mockito [Moc], Hamcrest [Ham] und Needle [Nee] verwendet. Bis auf Needle sollten alle genannten bekannt sein. Needle ist ein Framework zur Testunterstützung von Java EE-Anwendungen außerhalb eines Containers.

Wie es eingesetzt wird, sehen wir in Listing 8. Interessant ist dabei die Annotation `@Rule`, über die wir dem Test eine `NeedleRule` hinzufügen. Diese sorgt dafür, dass vor der Ausführung der Testmethode die mit `@ObjectUnderTest` und `@Inject` annotierten Felder der Klasse initialisiert werden. Das Objekt unter Test ist unser Präsentator. Needle analysiert die Klasse und erzeugt automatisch Mocks für Interfaces, die in der Klasse `ContactEditorPresenterBean` per `@Inject` verwendet werden. Über die Property `mock.provider` in der Datei `needle.properties` (muss im Klassenpfad liegen) legen wir fest, dass wir Mockito als Mock-Provider verwenden wollen.

Die eigentliche Testmethode ist nicht weiter spannend. Wir trainieren zunächst die bereitgestellten Mocks, führen die zu testende Methode des Präsentators aus und prüfen abschließend, ob unsere Erwartungen erfüllt sind.

```
public class ContactEditorPresenterTest {
  @Rule
  public NeedleRule rule = new NeedleRule();
  @Inject
  private ContactEditorView view;
  @Inject
  private ContactModel model;
  @Inject
  private Event<ContactEditingFinished> event;
  @ObjectUnderTest
  private ContactEditorPresenterBean presenter;
  @Test
  public void testApplyClicked() {
    Mockito.when(this.view.getFirstname()).thenReturn("First");
    Mockito.when(this.view.getLastname()).thenReturn("Last");
    Mockito.when(this.view.getEmail()).thenReturn("Email");
    Mockito.when(this.model.createOrUpdateContact(
      Mockito.anyLong(), Mockito.anyString(), Mockito.anyString(),
      Mockito.anyString())).thenReturn(Long.valueOf(1));
    this.presenter.buttonApplyClicked();
    Mockito.verify(this.model).createOrUpdateContact(null,
      "First", "Last", "Email");
    Mockito.verify(this.event).fire(Mockito.argThat(Matchers.allOf(
      Matchers.isA(ContactEditingFinished.class),
      Matchers.hasProperty("id", Matchers.equalTo(Long.valueOf(1))))));
    Mockito.verifyNoMoreInteractions(this.model, this.event);
  }
}
```

Listing 8: Testen eines Präsentators

Vaadin-CDI-Integration

Während der Entwicklung von Vaadin 7 wurde auch mit der Entwicklung einer erweiterten CDI-Integration [VaaCdi] von Vaadin begonnen. Im Wesentlichen wird ein neuer Scope `@UIScoped` bereitgestellt, der es erlaubt, den Lebenszyklus einer CDI-Bean an den Lebenszyklus eines Vaadin-UI zu binden. Dieser weicht vom Lebenszyklus einer HTTP-Session ab. So kann es mehrere UIs in einer HTTP-Session geben. Da sich das Projekt noch im Alpha-Stadium befindet, wird in der Demo-Anwendung zu diesem Artikel die Vaadin-CDI-Integration nicht verwendet. Im 2. Quartal 2014 soll die Version 1 erscheinen.

Literatur und Links

[DaBi] `http://en.wikipedia.org/wiki/UI_data_binding`
[DeAn] `https://github.com/akquinet/vaadin-cdi-mvp`
[Ham] `https://code.google.com/p/hamcrest/`
[JBoss] `http://www.jboss.org/jbossas/`
[Jun] `http://junit.org/`
[Mav] `http://maven.apache.org/`
[Moc] `https://code.google.com/p/mockito/`
[Nee] `http://needle.spree.de/`
[Obs] `http://de.wikipedia.org/wiki/Beobachter_(Entwurfsmuster)`
[PaVi] `http://www.martinfowler.com/eaaDev/PassiveScreen.html`
[SuCo] `http://www.martinfowler.com/eaaDev/SupervisingPresenter.html`
[VaaCdi] `https://github.com/vaadin/cdi`

Oliver Damm ist Berater und Softwarearchitekt bei der akquinet AG in Hamburg. In seinen fünfzehn Jahren Berufserfahrung hat er sich auf Kundenprojekte mit Java und JEE-Technologien spezialisiert. Seit 2010 beschäftigt er sich mit der Entwicklung von Webanwendungen unter Einsatz von Vaadin.
E-Mail: oliver.damm@akquinet.de