7. Umsetzung

> Clean code always looks like it was written by someone who cares.
>
> *(Michael Feathers)*

Inhaltsangabe

In diesem Kapitel werden die wichtigsten Aspekte der Realisierung erläutert. Zunächst wird die Architekturentscheidung begründet einen RESTful Webservice mit JSON zu konzipieren. Es folgt eine Überblick über die wichtigsten Funktionen des Webservice und die Schwierigkeiten bei der Deserialisierung.

Anschließend wird die Arbeitsweise der JSF-Komponente eines Dashboard-Items erläutert, welche für die Generierung des Einstellungsdialogs zuständig ist.

Im darauf folgenden Abschnitt wird auf die technische Umsetzung der Diagramm-Komponente eingegangen. Insbesondere die automatische Aktualisierung des Diagramms im Einstellungsdialog wird eingehend behandelt.

Zum Schluss wird erklärt welche Schritte aus Entwicklersicht notwendig sind um einen Rechenkern um Variabilität zu erweitern bzw. Varianten hinzuzufügen.

7.1. RESTful Webservice mit JSON

Für die Umsetzung der Webschnittstelle wurde ein RESTful Webservice konzipiert, der das Datenaustauschformat JSON benutzt.

Im Gegensatz zu SOAP ergeben sich folgende Vorteile:

- Dadurch, dass Nachrichten nicht in ein Envelope-Element eingebettet werden

müssen, ergeben sich weniger Verwaltungsdaten.

- Die Schnittstelle kann so konzipiert werden, dass die HTTP-Operationen GET, POST, PUT und DELETE semantisch der CRUD-Logik entsprechen. Die GET-Operation entspricht in diesem Fall dem Abruf einer Ressource und demnach einer Leseoperation. Bei SOAP läuft die Kommunikation ausschließlich über POST-Anfragen.
- Ressourcen des Webservices können allein durch die Angabe einer URI[1] abgerufen werden. Bei SOAP stehen die Beschreibungsparameter im POST-Body.
- Das Datenaustauschformat kann frei gewählt werden. Mit JSON gibt es kompakte und weit verbreitete Alternative zu XML. SOAP-Nachrichten sind immer im XML-Format.
  Es ist allerdings theoretisch möglich ein mit JSON serialisiertes Objekt per SOAP innerhalb der Nutzdaten zu übertragen.

Für die Konstruktion des Webservices war es notwendig eine Möglichkeit zu finden Java-Objekte ins JSON-Format umzuwandeln und umgekehrt. Aus diesem Grund wurden die Eigenschaften mehrerer JSON-Bibliotheken für Java untersucht, die den Entwickler bei diesen Aufgaben unterstützen. Die analysierten Bibliotheken unterscheiden sich zum Teil stark in ihrem Funktionsumfang. Folgende wichtige Bibliotheken, die jeweils unter einer freien Software-Lizenz stehen, sollen kurz vorgestellt werden:

json.org Hierbei handelt es sich um die Referenzimplementierung des Erfinders von JSON, Douglas Crockford. Der Funktionsumfang ist sehr elementar. Die Serialisierung komplexer Java-Objekte muss manuell durchgeführt werden, indem sukzessive ein Objekt vom Typ `JSONObject` aufgebaut wird. Analoges gilt für die Deserialisierung. Die Bibliothek wird nicht mehr weiterentwickelt.

Jackson Unterstützt dank *Data Binding*[2] die automatische Serialisierung und Deserialisierung komplexer Java-Objekte. Dabei wird standardmäßig auf die öffentlichen Zugriffsfunktionen des zu konvertierenden Objekts zugegriffen, welche Anhand ihrer Präfixe „get“ und „set“ identifiziert werden. Das serialisierte Objekt muss also nicht zwingend mit der internen Struktur des Ausgangsobjekts übereinstimmen. Des Weiteren kann der Serialisierungsprozess mit Hilfe von Annotationen angepasst werden. Die Bibliothek ist populär, weit verbreitet und wird aktiv weiterentwickelt.

Google GSON Unterstützt ebenfalls Data Binding. Greift bei der Umwandlung jedoch direkt auf die Attribute des zu serialisierenden Objekts zu (auch auf geschützte Attribute). Eine Anpassung des Serialisierungsprozesses kann entweder

[1] Uniform Resource Identifier
[2] Data Binding konvertiert JSON zu Java-Objekten und umgekehrt, basierend auf Zugriffsfunktionen, Attributen oder Annotationen.

über Annotationen erfolgen oder mit eigenen Java-Klassen, die das Interface `JsonSerializer` bzw. `JsonDeserializer` implementieren. Ein weiteres nützliches Feature ist die Unterstützung zur Umwandlung von Generics. Die Bibliothek ist populär und aufgrund ihrer geringen Größe von 190KB (v2.2.3) besonders im Mobile-Bereich weit verbreitet. Sie wird außerdem aktiv weiterentwickelt.

Für die Entwicklung der Webservice-Schnittstelle der Rechenkerne ist die Entscheidung zugunsten von *GSON* gefallen. Der Ansatz bei der automatischen De-/Serialisierung die geschützten Attribute mit zu berücksichtigen, schien subjektiv betrachtet zweckmäßiger. Außerdem ist es möglich mit geringen Anpassungen Generics zu verarbeiten, was sehr nützlich ist da das Variabilitätsmodell in Form des offenen Variabilitätspunktes aus Generics besteht.

Code-Listing 7.1 zeigt einen Ausschnitt der Implementierung der Java-Klasse `ExampleKernel`. Diese bildet den Webservice-Endpunkt und wird ein einem Webprojekt definiert. Man sieht die Methodendefinitionen von `getVariability`, `getDataSeries` und `getDataValue`.

```
1  @Stateless
2  @Path("example_kernel")
3  @Produces(MediaType.APPLICATION_JSON)
4  public class ExampleKernel {
5      @GET
6      @Path("variability")
7      public String getVariability() {
8        // (...)
9
10       // VariabilityModel serialisieren
11       return gson.toJson(model, VariabilityModel.class);
12     }
13
14     @GET
15     @Path("dataSeries/eom/{eom}/model/{model}")
16     public String getDataSeries(@PathParam("eom") String eom,
           @PathParam("model") String model) {
17       // VariabilityModel dekodieren und deserialisieren
18       // (...)
19
20       // DataSeries serialisieren
21       return gson.toJson(dataSeries, DataSeries.class);
22     }
23
24     @GET
25     @Path("dataValue/eom/{eom}/model/{model}")
26     public String getDataValue(@PathParam("eom") String eom,
           @PathParam("model") String model) {
27       // VariabilityModel dekodieren und deserialisieren
28       // (...)
```

```
29
30          // DataValue serialisieren
31          return gson.toJson(dataValue, DataValue.class);
32      }
33  }
```

Quelltext 7.1: Webservice Java-Code

Mit Hilfe von JAX-WS-Annotationen werden die Eigenschaften des Webservice definiert. Die Annotation `@Path` der Klasse spezifiziert den relativen Pfad der URL für den Webservice. `@Produces` definiert den Internet-Media-Type der HTTP-Antwort. In diesem Fall wird dieser auf `application/json` gesetzt.

Die Funktion `getVariability` definiert eine Ressource, die per HTTP-GET-Operation über folgende URL erreichbar ist:

`http://example.com/HelloWorld/rest/example_kernel/variability`

Die letzten zwei Segmente des Pfades entsprechen dem relativen Pfad, der durch die Annotationen konfiguriert wurde. Das Segment „HelloWorld“ bezeichnet den Namen der Java-EE-Anwendung und „rest“ wird per Servlet-Mapping in web.xml konfiguriert.

```
1  <servlet-mapping>
2    <servlet-name>Jersey REST Service</servlet-name>
3    <url-pattern>/rest/*</url-pattern>
4  </servlet-mapping>
```

Quelltext 7.2: web.xml

Der Rückgabewert ist die serialisierte Repräsentation des Variabilitätsmodells. Wie man in Zeile 11 von Listing 7.1 sieht, ist die Handhabung sehr leicht.

Interessanter ist der Fall für `getDataSeries`. Hier enthält der relative Pfad der `@Path`-Annotation dynamische Parameter, die von geschweiften Klammern umrandet sind. Diese werden in der Methodendeklaration via `@PathParam` aufgegriffen und in eine String-Variable injiziert.

Die Serialisierung mit GSON erfolgt weitestgehend automatisch. Erst wird eine GSON-Objekt erzeugt und anschließend die Methode `toJson` aufgerufen. Die Deserialisierung gestaltet sich aus zwei Gründen als etwas schwieriger.

1. Dadurch, dass die Klasse `VariationPoint` abstrakt ist, weiß GSON nicht welche Unterklasse zur Erzeugung des Objekts benutzt werden soll. Tatsächlich gibt es in Java keine effiziente und zuverlässige Vorgehensweise diese Information automatisch herauszufinden.
2. Der Typ der Klasse `OpenVariationPoint<T>` wird dynamisch festgelegt. Nach der Serialisierung geht die Typinformation jedoch verloren.

Damit der richtige Typ eines `OpenVariationPoint<T>`-Objekts bei der

Deserialisierung wiederhergestellt werden kann, wird die Typinformation in Form der String-Repräsentation des Package-Namespaces des parametrisierten Typs im Attribut *typeToken* gespeichert. Zusätzlich musste ein benutzerdefinierter Deserialisierer vom Typ `JsonDeserializer<VariationPoint>` implementiert werden, dessen Quellcode ausschnittsweise in Listing 7.3 zu sehen ist.

```
1  public class VariationPointDeserializer implements JsonDeserializer<
       VariationPoint> {
2      @Override
3      public VariationPoint deserialize(final JsonElement src, final
           Type srcType,
4          final JsonDeserializationContext context) throws
               JsonParseException {
5
6          JsonObject object = (JsonObject)src;
7
8          //...
9
10         // Distinguish between open and closed variation point
11         Object typeToken = context.deserialize(object.get("typeToken"
               ), String.class);
12         if(typeToken == null) {
13             // Instantiate ClosedVariationPoint
14
15             // ... assign fields
16
17             return variationPoint;
18         } else {
19             // Instantiate OpenVariationPoint<T>
20
21             // ... assign fields
22
23             return variationPoint;
24         }
25     }
26 }
```

Quelltext 7.3: VariationPointDeserializer.java

Anhand der Existenz des Attributs *typeToken* wird unterschieden, ob es sich um einen `ClosedVariationPoint` handelt oder einen `OpenVariationPoint`. Anschließend wird das entsprechende Objekt erzeugt und per Reflection die jeweiligen Parameter zugewiesen.

Der Quellcode der Klasse `VariabilityConstraint` ist in Listing 7.3 zu sehen.

```
1  public class VariabilityConstraint {
2      private transient Variant parentVariant;
3      private String parentId;
4      private transient Variant childVariant;
5      private String childId;
```

```
6       private VariabilityConstraintType type;
7
8       public VariabilityConstraint() {
9           super();
10      }
11
12      public VariabilityConstraint(Variant parentVariant, Variant
            childVariant, VariabilityConstraintType type) {
13          this.parentVariant = parentVariant;
14          this.childVariant = childVariant;
15          this.type = type;
16
17          // update special attributes for serialization!
18          this.parentId = parentVariant.getBelongsTo().getName() + ":"
                + parentVariant.getName();
19          this.childId = childVariant.getBelongsTo().getName() + ":" +
                childVariant.getName();
20      }
21
22      public Variant getParentVariant() {
23          return parentVariant;
24      }
25
26      public Variant getChildVariant() {
27          return childVariant;
28      }
29
30      public VariabilityConstraintType getType() {
31          return type;
32      }
33  }
```

Quelltext 7.4: VariabilityConstraint.java

Die Attribute *parentVariant* und *childVariant*, sind als *transient* deklariert und bleiben aufgrund dessen bei der Serialisierung unberücksichtigt. Stattdessen werden die speziellen Attribute *parentId* und *childId* serialisiert, die eine String-Konkatenation des dazugehörigen Variationspunktes, einem Doppelpunkt und der Variante sind. Bei der Deserialisierung muss dafür gesorgt werden, dass die transient-Attribute wiederhergestellt werden.

Abschließend bleibt festzuhalten, dass Java-EE ein gutes Framework für die Implementierung eines RESTful Webservice bietet. Unter Benutzung von funktionsreichen Bibliotheken wie Jackson oder GSON ist es auf einfache Art möglich komplexe Java-Klassen zu serialisieren und zu deserialisieren. Die Nutzung von abstrakten Klassen und Generics stellt ein Problem für die Deserialisierung dar, welches allerdings mit einem gewissen Mehraufwand gelöst werden kann.

7.2. Dashboard-Container als Composite Component

Für den Dashboard-Prototypen wurde eine Composite-Component entwickelt. Diese sorgt für die automatische Generierung des Einstellungsdialogs. Abbildung 7.1 zeigt einen Screenshot eines solchen Einstellungsdialogs.

Abbildung 7.1.: Einstellungsdialog Bildschirmfoto

Wie bereits im Konzept erläutert, ist der Dialog zweigeteilt. Auf der linken Seite befinden sich die Eingabefelder und auf der rechten Seite ist die Diagramm-Vorschau zu sehen, welche sich automatisch aktualisiert, wenn man auf der linken Seite die Einstellungen ändert. Alle Eingabefelder kommunizieren per Ajax mit dem Server und können eine Aktualisierung des Diagramms initiieren, ohne dass die gesamte Seite neu geladen wird.

Für geschlossene Variationspunkte werden entweder Auswahlfelder oder Gruppen von Checkboxen zur Mehrfachauswahl erzeugt. Für offene Variationspunkte werden Textfelder erzeugt, die dessen Validierung im JSF-Lebenszyklus durchgeführt wird (siehe Abschnitt 2.1.3). Abbildung 7.2b zeigt einen Validierungsfehler für einen offenen Variationspunkt vom Typ Integer. Aktuell werden fünf Datentypen von der JSF-Komponente unterstützt: Date, String, Integer, Float und Double.

Das Diagramm in Abbildung 7.1 enthält die Visualisierung für drei unterschiedliche Datenreihen, welche jeweils mit einem eigenen Variabilitätsmodell assoziiert sind. Für jede Datenreihe ergibt sich somit auch eine eigene Liste von Eingabefeldern. Zur besseren Organisation der Eingabefelder muss die zu bearbeitende Datenreihe explizit ausgewählt werden (siehe Abbildung 7.2a). Standardmäßig werden die Eingabefelder der ersten

(a) Datenreihe wählen (b) Validierungsfehler

Abbildung 7.2.: Weitere Funktionen des Einstellungsdialogs

Datenreihe angezeigt.

Weiterhin sind mehrere Dashboard-Items pro Dashboard-Container erlaubt. Für die Organisation mehrerer Dashboard-Items, wurde im Einstellungsdialog ein Akkordion-Menü implementiert. Die horizontal gestapelten Menüpunkte erlauben das Aufklappen jeweils eines Menüpunkts. Das Akkordion-Menü in 7.1 enthält nur den Menüpunkt „Ticket-ChartItem“.

Außerdem wurde ein Abbrechen-Mechanismus implementiert, um die Einstellungen zu verwerfen. Per Übernehmen-Schaltfläche, lassen sich die Einstellungen für das Dashboard übernehmen.

7.3. Diagramm-Rendering und Diagramm-Vorschau

Zur Darstellung der Diagramme wurde die Javascript-Bibliothek Highcharts benutzt.[3] Diese ist bereits fester Bestandteil der aktuellen MeDIC-Implementierung gewesen. Um eine automatische Diagramm-Vorschau zu realisieren, wurde eine neue Composite Component implementiert, welche besser wartbaren Code enthält als die aktuelle Implementierung und die Länge der XHTML-Ausgabe stark reduziert.

7.3.1. Aktuelle UIComponent-Implementierung

Die aktuelle Umsetzung der Diagramm-Komponente ist als Implementierung des UIComponent-Interfaces in Java umgesetzt. Diese kann, wie jede andere JSF-Komponenten (siehe Abschnitt 2.1.2 JSF), deklarativ in Facelets benutzt werden.

Für das Rendering muss die Methode `encodeBegin` implementiert werden, die die XHTML-Ausgabe generiert. Da Highcharts eine Javascript-Bibliothek ist und der richtige Rendering-Vorgang auf der Cient-Ebene stattfindet, wird an dieser Stelle ein leerer DIV-Block erzeugt und ein Block Inline-JavaScript erzeugt. Beim Client wird der Inline-JavScript-Code interpretiert und das Diagramm in den leeren DIV-Block

[3] http://www.highcharts.com/

gerendert.

Dieser Ansatz hat in seiner aktuellen Form zwei erhebliche Nachteile:

- Für jedes Dashboard-Item wird ein ca. 50 Zeilen langer Block Inline-JavaScript inmitten des XHTML-Markups erzeugt und die Ausgabe unnötig aufgebläht. Bei dem Beispiel aus Abbildung 3.1 sind zwölf Dashboard-Items zu sehen. Umgerechnet entspricht das 600 Zeilen zusätzlicher Ausgabe.
- Die Generierung des Inline-JavaScripts ist eine lange unübersichtliche Konkatenation von Zeichenketten und Variablen, welches sich negativ auf die Wartbarkeit und Erweiterbarkeit auswirkt.

7.3.2. Composite Component

Der neue Ansatz setzt auf ein deklaratives Konzept. Es wird kein Inline-JavaScript mehr erzeugt. Stattdessen werden die zur Erzeugung des Diagramms benötigten Variablen per HTML5-Data-Attribut an das DIV-Element angehängt. Obwohl diese Art von Attribut erst mit HTML5 spezifiziert wurde, ist die Browser-Unterstützung sehr gut.[4] Im Prinzip handelt es sich um ganz normale XHTML-Attribute mit dem Präfix „data-“.

Listing 7.5 zeigt eine Beispielausgabe der neuen JSF-Komponente für ein Dashboard-Item. Die serialisierten Datenreihen entsprechen denen des Diagramms aus Abbildung 7.1.

```
1 <div class="dashboard-item"
2     data-types="[&amp;quot;column&amp;quot;, &amp;quot;column&amp;quot;, &amp;quot;line&
      amp;quot;]"
3     data-names="[&amp;quot;Neue Tickets&amp;quot; ,&amp;quot;Geschlossene Tickets
      &amp;quot; ,&amp;quot;Offene Tickets&amp;quot;]"
4     data-values="[[1.0,8.0,10.0,7.0,12.0], [2.0,7.0,10.0,4.0,2.0],
      [2.0,4.0,4.0,5.0,10.0]]"
5     data-categories="[&amp;quot;Mo&amp;quot;, &amp;quot;Di&amp;quot;, &amp;quot;Mi&amp;quot;,
       &amp;quot;Do&amp;quot;, &amp;quot;Fr&amp;quot;]">
6     <!-- Hier wird das Diagramm beim Client gerendert. -->
7 </div>
```

Quelltext 7.5: Ausgabe der Composite Component

Das Auslesen und Interpretieren der Data-Attribute aller Dashboard-Items erfolgt über ein zusätzliches JavaScript, welches einmal im HEAD-Element der Seite eingebunden wird.

Listing 7.6 alle benötigten Schritte zur Initialisierung der Dashboard-Items.

```
1 $(function () {
2     // document ready event
```

[4] http://caniuse.com/#search=data

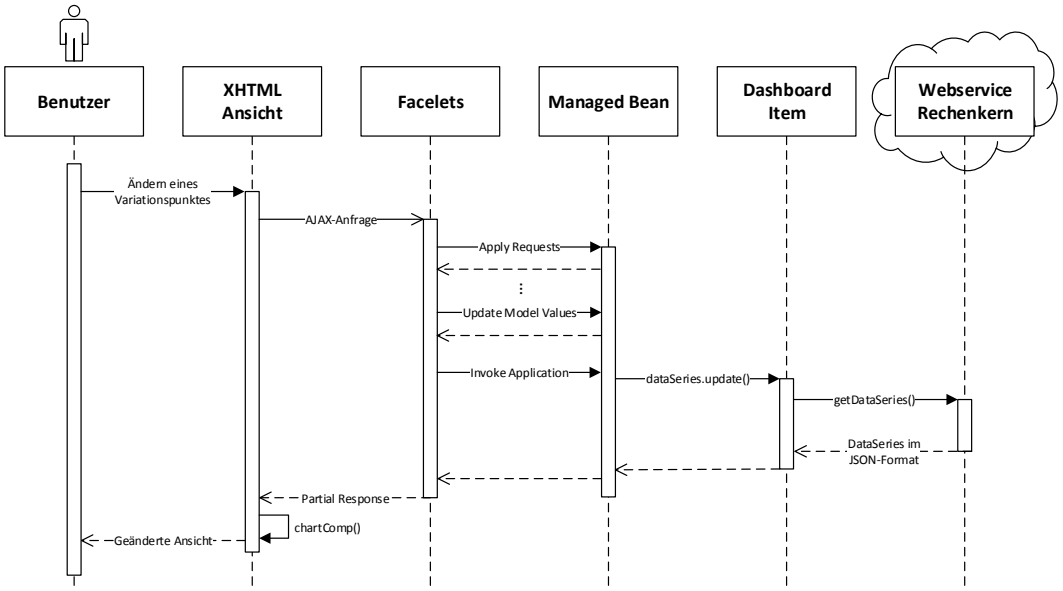

Abbildung 7.3.: Sequenzdiagramm zur asnychronen Aktualisierung des Diagramms

```
3
4       // initialize dashboard items
5       $('.dashboard-item').chartComp();
6   });
```

Quelltext 7.6: Document-Ready Ereignis und Initialisierung der Dashboard-Items

Die anonyme Funktion aus Zeile 1 wird ausgeführt, sobald das *Document Object Model* (DOM) der Seite vollständig geladen wurde. Die Schreibweise `$( ... );` ist die Kurzform des *documentReady*-Handlers von jQuery.[5] In Zeile 5 wird das jQuery-Objekt mit einem Selektor aufgerufen, der alle DOM-Elemente auswählt, die die Klasse *dashboard-item* haben. Anschließend wird auf diese Menge die Funktion `chartComp` angewendet, welche alle Dashboard-Items anhand ihrer Data-Attribute initialisiert.

7.3.3. Asynchrone Aktualisierung der Diagramm-Vorschau

Das Sequenzdiagramm in Abbildung 7.3 zeigt eine abstrahierte Darstellung der Aufrufreihenfolge, wenn der Benutzer im Einstellungsdialog einen Parameter ändert. Zunächst wird eine Ajax-Anfrage an den Server geschickt. Dort wird der JSF-Lebenszyklus durchlaufen (vgl. Abschnitt 2.1.3) und in der

[5] jQuery ist eine Javascript-Bibliothek, die in MeDIC für diverse Aufgaben benutzt wird.

Update-Model-Values-Phase schließlich das Variabilitätsmodell der entsprechenden Datenreihe aktualisiert. Anschließend wird die Update-Methode der Datenreihe aufgerufen, welche die Methode *getDataSeries* des dazu passenden Webservice mit dem aktualisierten Variabilitätsmodell aufruft. Im Reckenkern wird auf Basis der übermittelten Varianten und der EOM (Entity of Measurement) eine aktualisierte Metrik berechnet. Diese wird im JSON-Format serialisiert und zurückgegeben. Das Dashboard-Item speichert nun die neuen Werte.

In der letzten Phase des JSF-Lebenszyklus, wird eine *Partial Response* generiert, welche das aktualisierte Dashboard-Item enthält (siehe 7.5). Abschließend muss im Browser des Clients noch die Methode `chartComp` aufgerufen werden, um eine Neuinitialisierung des betreffenden Dashboard-Items in Gang zu setzen.

Folgendes Listing 7.7 zeigt das Ajax-Element für ein Eingabefeld der JSF-Komponente. Das Attribut *listener* definiert eine Methode im Managed-Bean, die während der Invoke-Applikation-Phase des JSF-Lebenszyklus aufgerufen werden soll. Die Methode *onModelChange* initiiert die Aktualisierung der Datenreihe.

Das Attribut *onevent* enthält eine Funktion, die für jedes Ereignis einer Ajax-Anfrage für dieses Element aufgerufen werden soll. Als Parameter erhält die Funktion die Variable *data*, die Informationen über das jeweilige Ajax-Ereignis enthält und die eindeutige ID der Diagramm-Vorschau innerhalb des XHTML-Dokuments.

```
1 <f:ajax event="valueChange"
2     execute="@this"
3     render="@this :#{helperBean.sanitizeId(cc.clientId)}:form:preview
          "
4     onevent="function(data){ $.chartEvent(data, '#{cc.clientId}
          :form:preview'); }"
5     listener="#{bean.onModelChange(dataSeries)}" />
```

Quelltext 7.7: Ajax-Element eines Variationspunktes

Listing 7.8 zeigt die Javascript-Funktion *chartEvent*, die für die Verarbeitung der unterschiedlichen Ajax-Ereignisse zuständig ist. Bei *begin*, wurde eine Ajax-Anfrage abgeschickt. Hier könnte dem Benutzer eine Ladeanimation gezeigt werden, damit er eine direkt Rückmeldung seiner Aktion bekommt. Das *complete*-Ereignis tritt ein, wenn die Antwort (Partial Response) empfangen wurde. Hier muss das die Kopie des alten Diagramm aus dem Speicher entfernt werden. Beim Ereigniss *success* kann schließlich mit der Methode *chartComp* das Diagramm mit den neuen Werten initialisiert werden.

```
1 $.chartEvent = function(data, id) {
2     // get the element of the diagram container
3     var element = document.getElementById(id);
4
5     if(data.status == 'begin') {
6         // show ajax loader
7     } else if(data.status == 'complete') {
8         // remove charts (if present) to purge memory
```

```
9           $(element).children().chartComp('destroy');
10      } else if(data.status == 'success') {
11          // create new charts for all children
12          // that match the selector
13          $(element).children('.dashboard-item').chartComp();
14      }
15  };
```

Quelltext 7.8: Ajax-Ereignisbehandlung zur Aktualisierung des Diagramms

7.4. Konfiguration der Rechenkerne aus Entwicklersicht

Aus Entwicklersicht gibt es zwei Schritte, die durchgeführt werden müssen, um einen Rechenkern um Variabilität zu erweitern oder Varianten hinzuzufügen.

1. Jeder Rechenkern besitzt eine Konfigurationsdatei, die eine Beschreibung der Variabilität enthält. Diese Beschreibung wird an die Visualisierungsschicht weitergereicht, um über die unterstützte Variabilität zu informieren. Hier muss der neue Variationspunkt eingetragen werden.

2. Als nächstes muss die Berechnungsvorschrift des Rechenkerns angepasst werden, sodass die neu hinzugefügten Varianten berücksichtigt werden. Die Metrik-Berechnung ist in Java-Code implementiert. Es muss also eine Anpassung des Java-Codes folgender Art erfolgen: „Falls Variante 1, dann tue dies und sonst das ... “

Die Konfigurationsdatei eines Rechenkerns ist eine statische Ressource, die pro Rechenkern mitinstalliert (deployed) wird. Der Inhalt ist die serialisierte Form einer Instanz von VariabilityModel im JSON-Format. Das Listing 7.9 zeigt ein Beispiel für den Rechenkern für die Zählung von Fehlertickets:

```
1  {
2    "varPoints": [
3      {
4          "typeToken": "java.util.Integer",
5          "selectedVariant": 10,
6          "name": "commentCount"
7      },
8      {
9        "variants": [
10         {
11           "name": "new",
12           "value": "new"
13         },
14         {
15           "name": "accepted",
16           "value": "accepted"
```

```
17          },
18          {
19            "name": "assigned",
20            "value": "assigned"
21          },
22          {
23            "name": "closed",
24            "value": "closed"
25          },
26          {
27            "name": "reopened",
28            "value": "reopened"
29          }
30        ],
31        "selectedVariantIndices": [
32          0,
33          1,
34          4
35        ],
36        "name": "status",
37        "multipleChoice": true
38      },
39      {
40        "variants": [
41          {
42            "name": "blocker",
43            "value": "blocker"
44          },
45          {
46            "name": "critical",
47            "value": "critical"
48          },
49          {
50            "name": "major",
51            "value": "major"
52          },
53          {
54            "name": "minor",
55            "value": "minor"
56          },
57          {
58            "name": "trivial",
59            "value": "trivial"
60          }
61        ],
62        "selectedVariantIndices": [
63          0,
64          1
65        ],
```

```
66        "name": "priority",
67        "multipleChoice": true
68      }
69    ],
70    "constraints": [
71      {
72        "parentId": "status:new",
73        "childId": "status:assigned",
74        "type": "REQUIRES"
75      },
76      {
77        "parentId": "priority:blocker",
78        "childId": "priority:major",
79        "type": "REQUIRES"
80      }
81    ]
82  }
```

Quelltext 7.9: Beispiel für eine Konfiguartionsdatei eines Variabilitätsmodells

Das dargestellte Variabilitätsmodell besteht aus drei Variationspunkten, *commentCount*, *status* und *priority*. Für *priority* gibt es fünf vordefinierte Varianten zur Auswahl. Die Standardvariante wird durch *selectedVariantIndices* angegeben und verweist auf die Elemente der Liste *variants* mit dem Index 0 und 1, *blocker* und *critical*. Bei *commentCount* handelt es sich um einen offenen Variationspunkt vom Typ Integer. Die Standardauswahl beträgt 10. Die Intention dieses Variationspunktes ist es, nur die Fehlertickets für die Zählung zu berücksichtigen, zu denen es 10 oder mehr Kommentare gibt. Ab Zeile 70 ist die serialisierte Darstellung von Beispiel-Beschränkungen (Constraints) zu sehen.