BUBE Online Support

Supportbericht 2012

Version 1.0 – 19.02.2013

Ansprechpartner
Christian Schneider
christian.schneider@quinscape.de
Telefon 0231 / 533 831 433
Telefax 0231 / 533 831 111

Inhalt

1 Einleitung

Dieser Bericht beschreibt die Tätigkeiten und Problemstellungen bei der Supportleistung für BUBE Online ab Version 1.1.32. Das Ziel des Dokuments ist eine Darstellung und kritische Reflexion der aktuellen Situation, um Schwachstellen sowohl in der Software als auch in der Projektdurchführung zu erkennen und bei Bedarf eine Optimierung herbeizuführen.

Als zentraler Ansprechpartner wurde uns Herr Schlinkmeier vom LANUV NRW zur Verfügung gestellt. Die Gesamtprojektleitung wurde von Frau Six vom UBA übernommen.

Das Management des Projekts wurde agil und kostensparend gehalten. Während des bisherigen Projektverlaufs war keine Eskalation, weder von durch den Auftraggeber, noch den Auftragnehmer, an Frau Six notwendig, so dass kein Aufwand für zusätzliche organisatorische Arbeitstreffen neben der Auftaktsitzung benötigt wurde.

2 Vorgehensweise und Aufwandsverteilung

Die Änderungsmeldungen wurden gemeinsam mit dem Auftraggeber priorisiert und in Meilensteinblöcke eingeteilt. Insbesondere wurden die Daten 31.05.2012 zur Veröffentlichung der XSD und 31.12.2012 zur Produktivstellung der neuen Version berücksichtigt.

Im Jahr 2012 wurden Leistungen in zwei Meilensteinen und eine Beauftragung einer Änderung durch Mecklenburg-Vorpommern erbracht, insgesamt ergab sich eine abrechnenbare Leistung von

Meilenstein 1:	64,25 Stunden
Meilenstein 2:	126,50 Stunden
Erweiterung Mecklenburg-Vorpommern:	75,50 Stunden

Von der Gesamtleistung von 266,25 Stunden entfallen insgesamt auf Projektmanagement und Auslieferung

17,50 Stunden (6,57%)

Der für Supportprojekte extrem niedrige Managementanteil begründet sich zum einen in einer schlanken Projektorganisation und der Nutzung eines webbasierten Changemanagementtools zur transparenten Dokumentation, als auch durch sehr kompetente Ansprechpartner des Auftraggebers mit hoher Einsatzbereitschaft, schneller Entscheidungsfindung und unkomplizierter Vorgehensweise. Insbesondere diese Unterstützung ermöglichte uns einen schnellen Einstieg in das System und die erfolgreiche Erbringung der Supportleistungen.

3 Weitere Planung

Bis Ende März 2013 ist ein dritter Meilenstein zur Umsetzung geplant. Die erwarteten Aufwände liegen hier zwischen 60 bis 170 Stunden. Die Varianz entsteht durch die Arbeitsweise, dass wir bei Änderungswünschen, deren Kosten/Nutzen-Verhältnis wir

nicht erkennen können, dies dem Auftraggeber zur Entscheidungsfindung vorlegen und diese zurückgestellt werden können.

4 Qualität der Software

Wir sind bestrebt durch unsere Leistungen unsere Auftraggeber zufrieden zu stellen und darauf aufbauend eine langfristige und vertrauensvolle Zusammenarbeit zu erzielen. In Hinblick auf diesen Beratungsanspruch und im Rahmen transparenter Arbeit möchten wir darauf hinweisen, dass die Software in dem uns vorliegendem Zustand einige Qualitätsprobleme aufweist. Meistens können wir diese bei Erweiterungen durch analog zum vorliegenden Programmcode durchgeführte Programmierung umgehen.

Das hat zum Ziel, dass wir die bestehenden Entwicklungsmuster[1] nicht brechen und die Software in ihren Grundzügen weiter der Möglichkeit einer Wartung unterliegt. Eine Modernisierung der Entwicklungsmuster würde je nach betroffenem Programmteil signifikante Änderungen und damit Aufwände, insbesondere für den Test, nach sich ziehen.

Demgegenüber birgt die Fortführung der bestehenden Entwicklungsmuster die Gefahr, dass zukünftige Erweiterungen entweder gar nicht oder nur mit hohem Aufwand durchzuführen sind. In den Unterkapiteln möchten wir Sie daher für einzelne Pogrammbereiche sensibilisieren, um Ihnen eine Orientierung der Möglichkeiten und Grenzen der Wartung in den kommenden Jahren zu geben.

Wir empfehlen dem Auftraggeber hier über mögliche Optimierungen, wie z.B. automatisierbare Testbarkeit, Inversion of Control (s.u.), Restrukturierung von komplexen Programmteilen, zur Modernisierung der Plattform zu diskutieren. Gerne stehen wir für eine solche Beratung im Rahmen des Supportsbudgets zur Verfügung.

4.1 Internationalisierung

Durch unsere Projektansprechpartner haben wir von dem Einsatz der Software in Mazedonien erfahren und damit von der Anforderung an die Internationalisierbarkeit der Software.

Bei der Umsetzung der Meilensteine entstand ein heterogenes Bild der Möglichkeiten innerhalb der Software. Generell sind die verwendeten Zeichenketten in einer Ressourcendatei durch Schlüssel abgebildet, so dass die meisten Beschriftungen und Meldungen internationalisiert werden können.

Leider ist dieser Ansatz nicht konsequent verfolgt worden, so dass sich neben den übersetzten Elementen an unterschiedlichen Stellen deutsche Meldungen sowohl auf der Oberfläche, in Reports, als auch bei Logausgaben ergeben.

Dies betrifft die Stellen:

---

[1] Die Erstellung von Software erfolgt i.d.R. nach bestimmten Mustern, um Wiederverwendbarkeit, Gleichförmigkeit und Wartbarkeit zu erreichen. Je nach Aufgabenstellung und Domäne der Software muss das geeignete Muster gewählt werden.

1. Javascript-Meldungen
   Die Meldungen in Javascriptanteilen (z.B. „Möchten Sie die Daten löschen“) sind als Texte fest in die Javascripte integriert und müssen individuell übersetzt werden.
2. XSLT-Reports
   Die Beschriftungen und Meldungen bei PDF-Reports (z.B. PRTR) sind in deutscher Sprache fest in die Templates integriert und müssen individuell übersetzt werden.
3. Logausgaben
   In den Logausgaben werden deutsche Begriffe wie „Fehler“ benutzt.

Insbesondere Punkt 1 und 2 führen bei einer individuellen Übersetzung zu einer Abspaltung der übersetzten Software und müssen in dieser separat gewartet werden. Insbesondere Änderungen an den Oberflächenmasken und Reports können nicht mehr von aktualisierten Versionen übernommen werden und müssen separat migriert werden. Aus der Erfahrung heraus werden diese Migrationsaufwände nicht gering sein und führen in der Regel zur Einstellung der Wartung.

Die Logausgaben könnten mit mäßigem Aufwand in englische Sprache übersetzt werden, da sie i.d.R. nur durch technisch geschultes Personal gelesen werden.

Ausgehend von diesen Punkten ist BUBE Online zum jetzigen Zeitpunkt in Hinblick auf die Weiterentwicklung und Wartung nur für eine Ausführung in deutscher Sprache geeignet.

4.2 Fehlende Testfälle

Die Übergabe des Projekts erfolgte ohne Testspezifikationen oder automatisierte Tests. Innerhalb des Projekts muss die fehlende Testbarkeit durch Klicktests kompensiert werden, die sowohl vom Auftraggeber, als auch vom Auftragnehmer durchgeführt werden.

Ein Regressionstest[2] ist unter diesen Umständen nicht möglich, sondern nur ein begrenzter Modultest. Dies kann über eine längere Zeit zu Qualitätseinbußen des Gesamtsystems führen, die zu weiteren Aufwänden zur Beseitigung der Nebeneffekte führen.

Ein Regressionstest wird in der Regel als *automatisierter* Test ausgeführt, da durch die Ausführung als Klicktest signifikante Aufwände entstehen, die häufig nicht in Relation zum Umfang einer Änderungen stehen (Ergänzung einer Fallunterscheidung hat ggf. Einfluss auf andere Module, ein 2-tägiger Regressionstest als Klicktest aller Module ist jedoch nicht darstellbar).

Für die Erstellung *automatisierter* Regressionstest ist jedoch eine darauf optimierte Strukturierung des Quellcodes bzw. der Plattform notwendig, die insbesondere zwei Bedingungen erfüllen muss:

[2] http://de.wikipedia.org/wiki/Regressionstest

- Quellcode-Modularisierung
  Die Module oder Services eines Systems sollten in Einheiten, den Units, zerlegt sein und per Inversion of Control[3] zusammengesetzt werden. Damit lassen sich die so entstandenen Units isoliert testen und bilden das Fundament der Regressionstests.

- Identifizierbare Oberflächenkomponenten
  Um die Kernprozesse an der Oberfläche zu testen bedarf es der Möglichkeit einer eindeutigen Identifikation der Elemente. Ist diese gegeben, lässt sich durch Frameworks wie WebDriver[4] eine thematisch angepasste Testsprache (DSL – Domain Specific Language) erzeugen. Mit Unterstützung einer solchen DSL lassen einfach und effzient Oberflächentests formulieren, die über WebDriver auf unterschiedlichen Browser automatisiert ausgeführt werden können.

Beide Voraussetzungen sind bei BUBE Online nicht gegeben, so dass automatisierte Regressionstests momentan nur sehr aufwändig realisierbar wären.

Der Quellcode ist sehr redundant aufgebaut, d.h. für die gleiche Logik müssen mehrfach die gleichen Tests geschrieben werden. Jeder einzelne dieser Tests muss bei Änderung der Logik entsprechend redundant angepasst werden.

Die Oberfläche selbst folgt keinen eindeutigen Regeln, so dass automatisierte Oberflächentests sehr individuell umsetzbar sind, was den entsprechenden Aufwand nach sich zieht, da viele Elemente einzeln identifiziert werden müssen.

Wir sehen automatisierte Tests als ein wichtiges Element zur Qualitätserhaltung, insbesondere in den kommenden Jahren der Pflege. Gerne beraten wir im Rahmen des Supportbudgets.

4.3 Codequalität

Die Codequalität und die eingesetzten Entwicklungsmuster von BUBE Online entsprechen durchaus dem Alter des Systems. In Hinblick auf die Wartung des Systems sind bei der Erstellung der Software einige Entscheidungen getroffen worden, die sich ungünstig auf bestimmte Arten von Änderungswünschen auswirken. Wir möchten diese hier mit Beispielen darstellen, um Ihnen zu ermöglichen unsere Aufwandsschätzungen nachvollziehen zu können.

Zudem sollte auch hier ein Lösungsansatz diskutiert werden. Wir können im Rahmen des Supportbudgets eine tiefergehende Analyse anbieten.

4.3.1 Code-Redundanz

Das System ist geprägt von starker Redundanz über die implementierenden Klassen, insbesondere der Action-Klassen. Zum Teil ist die gleiche Logik mehrfach an unter-

[3] http://de.wikipedia.org/wiki/Inversion_of_Control

[4] http://google-opensource.blogspot.de/2009/05/introducing-webdriver.html

schiedlichen Stellen implementiert. Als Beispiel dient die Funktion der Selektion aller Einträge einer Listenseite:

```
if (action.equals(messages.getMessage("button.selectAll"))) {
    if (listBean.getListe() != null) {
        wahl = new String[listBean.getListe().length];
        for (int i = 0; i < listBean.getListe().length; i++) {
           if (listBean.getListe()[i] != null) {
               wahl[i] = String.valueOf(listBean.getListe()[i].getStoffnr());
           }
       }
       if (wahl != null) {
          dForm.set("c_wahl", wahl);
       }
    }
   […]
   return toInputPage(mapping, form, request, true, false, true, formValue);
```

Diese Logik wiederholt sich für jede Listenseite im System, obwohl nur eine Variable (der Primärschlüssel der Liste, hier die Stoffnummer) sich ändert. Würde sich im Laufe der Wartung die Methodik zur Selektion ändern, müssten die Aufwände für die Änderung mit der Anzahl der Listenmasken multipliziert werden. Analog gibt es weitere Beispiele für Detailmasken und Funktionsseiten.

Ein weiteres Beispiel zeigt die Auswirkung konkret an Bug 261 (Desktop-Navigation für GFA). In diesem Fall ist die Logik zur Darstellung der Desktopelemente vollständig in Fallunterscheidungen untergebracht, wie ein verkleinerter Ausschnitt aus der Klasse DynaNav zeigt:

```
       if  (openNodeTemp.startsWith(DynaNavTree.SPECIAL_IDENTIFIER_1_0_MODUL_E11)
|| openNodeTemp.endsWith(DynaNavTree.SPECIAL_IDENTIFIER_1_0_MODUL_E11)) {
               loadListE_anlage = true;
       }
       if  (openNodeTemp.startsWith(DynaNavTree.SPECIAL_IDENTIFIER_2_0_MODUL_E11)
|| openNodeTemp.endsWith(DynaNavTree.SPECIAL_IDENTIFIER_2_0_MODUL_E11)) {
               loadListE_an = true;
               loadListE_anlage = true;
       }
       if  (openNodeTemp.startsWith(DynaNavTree.SPECIAL_IDENTIFIER_3_0_MODUL_E11)
|| openNodeTemp.endsWith(DynaNavTree.SPECIAL_IDENTIFIER_3_0_MODUL_E11)) {
               loadListE_quelle = true;
       }
       if                                                                (open-
NodeTemp.startsWith(DynaNavTree.SPECIAL_IDENTIFIER_3_0_E_ANLAGE_MODUL_E11)     ||
openNodeTemp.endsWith(DynaNavTree.SPECIAL_IDENTIFIER_3_0_E_ANLAGE_MODUL_E11)) {
               loadListE_quelle_e_anlage = true;
               loadListE_anlage = true;
       }
       if  (openNodeTemp.startsWith(DynaNavTree.SPECIAL_IDENTIFIER_4_0_MODUL_E11)
|| openNodeTemp.endsWith(DynaNavTree.SPECIAL_IDENTIFIER_4_0_MODUL_E11)) {
```

```
                loadListE_ghs = true;
                loadListE_anlage = true;
        }
        if  (openNodeTemp.startsWith(DynaNavTree.SPECIAL_IDENTIFIER_5_0_MODUL_E11)
|| openNodeTemp.endsWith(DynaNavTree.SPECIAL_IDENTIFIER_5_0_MODUL_E11)) {
                loadListE_brenn = true;
                loadListE_ghs = true;
                loadListE_anlage = true;
        }
        if  (openNodeTemp.startsWith(DynaNavTree.SPECIAL_IDENTIFIER_6_0_MODUL_E11)
|| openNodeTemp.endsWith(DynaNavTree.SPECIAL_IDENTIFIER_6_0_MODUL_E11)) {
                loadListE_tier = true;
                loadListE_ghs = true;
                loadListE_anlage = true;
        }
        if                                                              (open-
NodeTemp.startsWith(DynaNavTree.SPECIAL_IDENTIFIER_4_0_E_AN_MODUL_E11)  ||  open-
NodeTemp.endsWith(DynaNavTree.SPECIAL_IDENTIFIER_4_0_E_AN_MODUL_E11)) {
                loadListE_ghs_e_an = true;
                loadListE_an = true;
                loadListE_anlage = true;
        }
        if                                                              (open-
NodeTemp.startsWith(DynaNavTree.SPECIAL_IDENTIFIER_5_0_E_AN_MODUL_E11)  ||  open-
NodeTemp.endsWith(DynaNavTree.SPECIAL_IDENTIFIER_5_0_E_AN_MODUL_E11)) {
                loadListE_brenn_e_an = true;
                loadListE_ghs_e_an = true;
                loadListE_an = true;
                loadListE_anlage = true;
        }
```

Eine Umsetzung auf Basis dieser Logik würde mehrere Tage in Anspruch nehmen, da die fehlende Übersicht und starke Verschachtelung in der Klasse eine sehr hohe Aufmerksamkeit bei Änderungen und damit eine langsame Arbeitsgeschwindigkeit bedingt. Zudem sind sehr viele Testläufe notwendig sind, um sicherzustellen, dass die ergänzte Logik nicht mit der bestehenden Logik interferiert. Insgesamt hat die Klasse über 4200 Zeilen, davon entfällt mehr als die Hälfte, ca. 2260 Zeilen, auf die Erstellungslogik des Desktopbaums (Methode getTree), die für jedes Modul eine Steuerlogik per Fallunterscheidung berechnet (s.o.).

Damit kumulieren sich durch fehlende Modularisierung die Aufwände, die sich aufgrund der Codestruktur ergeben für eine verhältnismäßig einfache Anforderung auf mehrere Tage.

Aus der redundanten Auslegung von einigen Codebereichen ergeben sich somit Systembereiche, die nur mit verhältnismäßig hohem Aufwand änderbar sind.

4.3.2 Kombinierte Logik

In vielen Objekt-Beans befindet sich kombinierte Logik, die teilweise Polymorphismus simuliert statt die JAVA inhärenten Mechanismen zu nutzen. Dies führt zu schwer nachvollziehbarem Verhalten und damit zusätzlichen potentiellen Fehlerquellen:

eEvBean.loadEintraege(anlRefkey, false, anlntkey , sess);

Diese Funktion lädt alle emissionsverursachenden Vorgänge einer Anlage anlRefkey inklusive aller Anlagenteile. Übergibt man einen Primärschlüssel eines Anlagenteils anlntkey, so werden nur die emissionsverursachenden Vorgänge des Anlagenteils einer Anlage geladen, aber nur, wenn das übergebene Flag ‚false' ist, ist es dagegen ‚true', dann wird der Primärschlüssel des Anlagenteils ignoriert und grundsätzlich alle emissionsverursachenden Vorgänge geladen.

Diese Art der kombinierten Logik über Flags führt neben der sehr schlechten Lesbarkeit dazu, dass die anhand der Methodensignatur die Funktionsweise nicht deutlich zu erkennen ist. Damit ist zusätzlicher Aufwand bei der Entwicklung zur Prüfung der Funktionsweise notwendig.

Weitere Beispiele zeigen, dass Signaturen teilweise wesentlich mehr Flags beinhalten. An dem folgenden Beispiel kann man sehr schön erkennen, dass insbesondere bei booleschen Flags, diese nur durch Abzählen zu differenzieren sind:

```
xml1Bean.exportEintraege(wahlLong, Constants.MODUL_E11, true, false, false, false, false, false, false, false, false, false, false, true, false, userObject);
```

Die Flags selbst dienen zur Steuerung einer ca. 590 Zeilen langen Methode, in der diese nach und nach zur Fallunterscheidung genutzt werden.

Diese Beispiele machen deutlich, dass einige Teile des Systems nur sehr aufwändig bzw. sehr gründlich und damit aufwandsintensiv gewartet werden können. Wir bitten Sie diese Situation bei der Diskussion in Bezug auf das zukünftige Supportbudget zu berücksichtigen.

4.4 Datenvalidierung

Die Validierung der Datenfelder wurde über mehrere Indirektionen modelliert, die entsprechend korrekt verbunden werden müssen. Die Validierungen sind über Referenzschlüssel in der Datei validation_grid und zum Großteil redundant referenziert in der Datei validation.xsl abgelegt. Aus diesen beiden Dateien wird dann die Datei validation.xml berechnet.

Durch die Dezentralisierung müssen die korrekten Schlüsselwerte eingetragen werden. Die Datei validation_grid.xml ist mittlerweile über 13.000 Zeilen groß, die zugehörige xsl-Datei über 1.100 Zeilen:

validation_grid.xml:

```
[…]
  <form name="sarbFormselectLoeschen">
```

```
    <field property="c_wahl" depends="required">
     <msg name="required" key="errors.required" />
     <arg0 key="sarb.list.ch1" />
    </field>
    <field property="throbber" />
  </form>
[…]
```

validation.xsl:

```
[…]
    <form name="sarbFormselectLoeschen">
      <xsl:copy-of select="//formset/form[@name='sarbFormselectLoeschen']/field"/>
    </form>
[…]
```

In diesem Beispiel setzt sich die Regel aus dem Maskennamen “sarbForm” und der jsAction „selectLoeschen“ zusammen. Diese muss identisch in der validation.xsl eingetragen werden und über den xpath-Selector entsprechend referenziert werden. Damit wird dezentral drei Mal auf den Formnamen verwiesen, dies wiederum multipliziert mit den möglichen Anwendungsfällen, wie z.B. Prüfung, Export u.a.

Durch dieses Konzept ist eine Umbenennung, z.B. aufgrund einer Erweiterung durch eine Zwischenmaske nur aufwändig möglich. Daher sollte – unabhängig von der fachlichen Abbildung – eine Veränderung bestehender Prozesse, die Änderungen hier notwendig machen, in Zukunft sehr genau geprüft werden.

Für die Ergänzung bestehender Masken muss anhand von Benennungsregeln innerhalb der 13.000 Zeilen die korrekte Stelle gefunden und angepasst werden. Eine Prüfung auf Korrektheit der Änderungen kann nur indirekt durch einen Klicktest geprüft werden, da die Regeln zentral verwaltet und bei Neustart der Applikation aktualisiert werden.

Die Erstellung einer neuen Maske ist entsprechend aufwändiger, da hier redundante Informationen in unterschiedlichen Dateien gepflegt werden müssen. Insbesondere die Indirektion durch Schlüssel, die durch Benennungsregeln erzeugt werden, erfordert eine hohe Sorgfalt bei der Prüfung der Änderungen und damit entsprechenden Aufwand.

Der Validierungsmechanismus wächst aufgrund seiner Zentralisierung stetig an, was zukünftige Änderungen zunehmend komplexer machen wird.

4.5 BITV2-Kompatibilität

Die Applikation ist prinzipiell mit der BITV2 kompatibel. Dabei ist zu beachten, dass keine Kompatibilitätsschicht zwischen den Oberflächenelementen und den Anforderungen der BITV2 existiert.

Dies bedeutet, dass Anforderungen an die BITV2 nicht automatisch durch ein Framework erfüllt werden, sondern individuell für neue Elemente implementiert werden müssen. Dieser Mehraufwand hält sich zum jetzigen Zeitpunkt in Grenzen, so dass hier kein unmittelbarer Handlungsbedarf entsteht.

5 Fazit

Insgesamt konnten wir im Rahmen der Supportaufgaben mit einem produktiv einsetzbaren Softwaresystem arbeiten. Die innere Struktur von BUBE Online ist gleichförmig über die Module und erlaubt so eine gute Orientierung innerhalb der bestehenden Strukturen. Dank der sehr guten Unterstützung des Auftraggebers konnten Probleme frühzeitig erkannt und behoben werden. Somit waren wir in der Lage effizient unsere Supportleistungen zu erbringen.

Aus aktueller Sicht ist die Erhaltung des Status quo des Softwaresystems problemlos weiterhin mit moderaten Pflegeaufwänden möglich.

Ist für zukünftige Versionen mit neuen oder steigenden Anforderungen an die Prozesse zu rechnen, so empfehlen wir aufgrund der aufgezeigten Schwächen des Systems einen erhöhten Pflegeaufwand einzuplanen.

Die Architektur des Systems entspricht – unter Berücksichtigung des Alters der Software – nicht mehr modernen Anforderungen des Software-Engineerings. Insbesondere die fehlende Möglichkeit für die Erstellung automatisierter Regressionstest wird sich zukünftig stärker bemerkbar machen.

Die Herausforderung für die Pflege der Software in den kommenden Jahren wird eine Abwägung der Erhaltung des Status quo und der maßvollen Veränderung der Portalfunktionen sein, die sich wirtschaftlich vertreten lässt. Bei der Bewertung dieser Schritte stehen wir Ihnen gerne als Partner zur Seite und hoffen, dass wir bis jetzt unserem Beratungsanspruch gerecht geworden sind.