Informatik-A

Design and Java implementation of a test data generator for e-books

Kai Weber

Bachelor-/Master-Abschlussarbeit

Betreuer: Prof. Dr. Andreas Lux

Stuttgart, den 12.3.2016

Kurzfassung

In der vorliegenden Arbeit wird die Umsetzung eines kleinen Hilfsprogramms für Software-Abnahmetests vorgestellt. Mit diesem Hilfsprogramm werden Testdateien erzeugt, welche die Personen, die Abnahmetests durchführen, unmittelbar verwenden können.

Bei dem zu testenden System handelt es sich um eine digitale Verlagsauslieferung. Die für die Tests benötigten Dateien sind E-Book-Dateien, Metadatendateien und Grafikdateien. Das Testteam kann im Hilfsprogramm bestimmte Ausprägungen der benötigten Testdateien in einer grafischen Benutzeroberfläche konfigurieren und die Daten so erzeugen.

Die Umsetzung dieses Hilfsprogramms („Testdatengenerator“) erfolgte in Java. Bei der Modellierung wurden insbesondere die Entwurfsmuster „Erbauer“ und „Beobachter“ verwendet; eine Model-View-Controller-Struktur wurde hingegen nicht durchgängig, sondern nur sehr punktuell angewandt.

Das entwickelte Programm wurde bereits während der Umsetzung im praktischen Einsatz erprobt und erwies sich beim Lösen einiger zuvor bestehender Probleme im Umgang mit Testdaten als hilfreich. Die Konzeption, Umsetzung und Erprobung erfolgte im Zeitraum von ca. sieben Monaten, wobei nur eine Auswahl der sinnvollen Konfigurationsmöglichkeiten verwirklicht werden konnten. Eine Liste der noch offenen Wünsche an Funktionalität und Codestruktur ist ebenfalls Bestandteil dieser Übersicht.

Abstract

This work presented here presents the implementation of a little software tool for facilitating user’s acceptance testing. This tool generates test files which can be directly used by the acceptance testing team.

The system which is to be tested is a distribution system for electronic publications. The files required by the acceptance testing team are e-books, metadata files and images. The testing team can configure certain aspects of these required files in a graphical user interface before actually generating these files.

The implementation of this tool (“test data generator”) was done in Java. In the modelling of the classes the design patterns “builder” and “observer” were primarily used. A “model-view-controller”-structure was not used throughout, but only partially, were it deemed helpful.

During the implementation phase the test data generator program was already used in practice and proved to be helpful when for the handling of test data. The design, implementation and testing of the tool took around 7 months. During this period only a subset of useful configuration options could be implemented. A list of desiderata concerning functionality and code structure is part of this work presented here.

Inhaltsverzeichnis

[1 Einleitung und Problemstellung 1](#_Toc445552531)

[1.1 Ausgangslage 1](#_Toc445552532)

[1.2 Projektziel 3](#_Toc445552533)

[1.3 Übersicht über diese Projektdokumentation 3](#_Toc445552534)

[2 Beschreibung der zu erzeugenden Dateiformate 4](#_Toc445552535)

[2.1 ONIX 4](#_Toc445552536)

[2.2 E-Book-Dateien 5](#_Toc445552537)

[2.3 Grafik-Dateien 6](#_Toc445552538)

[3 Kurzbeschreibung der grafischen Benutzeroberfläche 6](#_Toc445552539)

[4 Architekturübersicht 8](#_Toc445552540)

[4.1 Die Pakete des Testdatengenerator-Quellcodes 9](#_Toc445552541)

[4.1.1 testdatagen.model 9](#_Toc445552542)

[4.1.2 testdatagen.model.files 11](#_Toc445552543)

[4.1.3 testdatagen.onixbuilder 12](#_Toc445552544)

[4.1.4 testdatagen.gui 14](#_Toc445552545)

[4.1.5 testdatagen.gui.listeners 14](#_Toc445552546)

[4.1.6 Restliche Pakete 14](#_Toc445552547)

[4.2 Verwendete externe Bibliotheken 15](#_Toc445552548)

[4.3 Codestil-Konventionen 15](#_Toc445552549)

[5 Zusammenfassung und Ausblick 16](#_Toc445552550)

[5.1 Ausblick 1: Ausbau fachlicher Funktionalität 16](#_Toc445552551)

[5.2 Ausblick 2: Verbesserung der Codestruktur und der Entwicklungs-Flexibilität 17](#_Toc445552552)

[Index 19](#_Toc445552553)

[Glossar 20](#_Toc445552554)

[Anhang: Übersicht über die Inhalte der beiliegenden CD-ROM 21](#_Toc445552555)

[Erklärung der Kandidatin / des Kandidaten 22](#_Toc445552556)

Abbildungsverzeichnis

[Abbildung 1: ONIX-Formate - Beispiel eines Knotens mit Preisinformationen 4](#_Toc445552276)

[Abbildung 2: Das Hauptfenster mit den Beispielszenarien 7](#_Toc445552277)

[Abbildung 3: Bearbeitungsansicht eines Testszenarios 7](#_Toc445552278)

[Abbildung 4: Detailkonfiguration eines Produkts 8](#_Toc445552279)

[Abbildung 5: Zentrale Modellklassen 10](#_Toc445552280)

[Abbildung 6: Das Dateimodell 11](#_Toc445552281)

[Abbildung 7: Title und Files 12](#_Toc445552282)

[Abbildung 8: Die OnixPartsBuilder-Klasse 13](#_Toc445552283)

[Abbildung 9: OnixPartsDirector 13](#_Toc445552284)

# Einleitung und Problemstellung

## Ausgangslage

Die Koch, Neff & Oetinger Verlagsauslieferung GmbH (kurz KNO-VA) entwickelt und betreibt eine Software „E-Book-Plant“ zur Verwaltung und Auslieferung von E-Books und verwandten Produkten (elektronischen Hörbüchern, Software-Downloads). Verlage liefern diese elektronischen Produkte, Metadaten und „Zubehör“ (z.B. Coverabbildungen) an die E-Book-Plant, wo auf Basis von hinterlegten Regeln und Konfigurationen Qualitätskontrollen stattfinden, bevor die Produkte dann an Verkaufsstellen („Retailer“) ausgeliefert werden. Der gesamte Vorgang wird bei der KNO-VA und insgesamt in der Buchbranche als „digitale Verlagsauslieferung“ bezeichnet - in Anlehnung an den traditionellen Begriff der Verlagsauslieferung, der die Lagerung und Fakturierung physischer Verlagsprodukte betrifft.

Die E-Book-Plant wird dabei von Mitarbeiterinnen und Mitarbeitern der KNO-VA (kurz DiVA-Team) bedient, indem im Austausch mit Verlagen und Retailern Regeln abgesprochen und dann durch das DiVA-Team in der E-Book-Plant konfiguriert werden. Externe, also nicht bei der KNO-VA angestellte Personen, erhalten keinen direkten Zugriff auf die E-Book-Plant; jedoch überwacht die E-Book-Plant Ordner eines FTP-Servers, in welchem die Verlage Dateien bereitstellen. Die in diesen Ordnern bereitgestellten Produktdaten werden dann auf Basis der Konfigurationen automatisiert verarbeitet und an die angeschlossenen Retailer verteilt. Ein manuelles Eingreifen in die Distributionsprozesse durch das DiVA-Team ist normalerweise nur in Fehlerfällen nötig.

Zur Verdeutlichung der Abläufe in der E-Book-Plant hier ein typisches Beispiel: Der Verlag V sendet zwei Monate vor dem Erscheinungstermin ein neues E-Book in den FTP-Order der E-Book-Plant. Die E-Book-Plant weiß, dass alle Produkte des Verlags V an die Retailer A, B und C ausgeliefert werden müssen. Retailer A erhält dabei stets das Produkt (d.h. eine E-Book-Datei), die Metadaten und eine Coverabbildung im Format 500 x 700 Pixel, sobald diese Daten vorliegen. Retailer B erhält dieselben Daten, jedoch grundsätzlich immer erst 14 Tage vor dem Erscheinungstermin. Retailer C benötigt neben dem Cover in der Größe 600 x 840 Pixel auch zwingend eine Preisangabe in Schweizer Franken. Die E-Book-Plant skaliert nun die vom Verlag gelieferten Coverdateien in die gewünschten Zielgrößen und liefert die Daten zu den konfigurierten Terminen aus. Sie prüft im Falle von Retailer C des Weiteren, ob tatsächlich vom Verlag ein Schweizer Preis mitgeteilt wurde. Ist dies nicht der Fall, gibt sie eine Fehlermeldung aus, woraufhin das DiVA-Team tätig wird und eine Schweizer Preisangabe beim Verlag anfordert. Die wesentlichen Aufgaben der E-Book-Plant bestehen somit in der Validierung von Metadaten (z.B. „Ist ein Schweizer Preis vorhanden, wo nötig?“) sowie in der Modifikation von Daten (z.B. Skalieren von Grafikdateien).

Aus dem Markt heraus ergeben sich immer wieder neue Anforderungen, die zur Notwendigkeit von Anpassungen am Programmcode der E-Book-Plant führen. So müssen etwa neue Dateiformate unterstützt werden, neue Konfigurations- oder Validierungsmöglichkeiten geschaffen werden, neue statistische Auswertungen ermöglicht werden, neue Plausibilitäts- oder Integritätsprüfungen vorgesehen werden. Die Weiterentwicklung der E-Book-Plant erfolgt mit agilen Methoden, so dass kurze Releasezyklen (in der Regel alle 5 Wochen) vorherrschen. Vor Releases werden sowohl Abnahmetests der neuen Funktionen als auch manuelle Regressionstests durchgeführt. Während die Programmierer[[1]](#footnote-1) für das Schreiben und Durchführen automatisierter Unit-Tests und Integrationstests verantwortlich sind, obliegt die Durchführung von Abnahme- und Regressionstests dem DiVA-Team. Hierbei geht es sowohl um die Überprüfung, ob eine gewünschte Funktionalität korrekt spezifiziert und umgesetzt wurde als auch darum sicherzustellen, dass durch Programmänderungen keine vorhandene Funktionalität verfälscht oder anderweitig beeinträchtigt wurde.

Zur Durchführung solcher Anwenderinnentests[[2]](#footnote-2) benötigt das DiVA-Team Testdaten, die je nach Testfall unterschiedlichen Anforderungen genügen müssen. Einerseits sollen die Testdaten in formaler Hinsicht - insbesondere was die Dateiformate betrifft - den üblicherweise von Verlagen angelieferten Daten entsprechen, andererseits sollen bestimmte Ausprägungen der Testdaten gezielt beeinflusst werden können. Hierzu ein einfaches Beispiel: Produktmetadaten werden üblicherweise in einem XML-Format namens ONIX geliefert (s.u. Kapitel 2.1). Innerhalb einer ONIX-Datei können unterschiedliche Produktpreise, z.B. in verschiedenen Währungen oder für bestimmte Länder oder Zwecke gemeldet werden. In der Regel genügt es der E-Book-Plant, wenn mindestens ein deutscher Europreis in den Produktdaten vorhanden ist. Eine zu testende neue Funktionalität könnte sich aber auf die Verarbeitung von Produktdaten beziehen, in denen auch US-Dollar-Preise vorkommen. Um zu Testzwecken an eine ONIX-Datei mit entsprechend passenden Preisen zu gelangen, gibt es prinzipiell drei denkbare Möglichkeiten:

1. Das DiVA-Team nutzt Suchmöglichkeiten der E-Book-Plant oder von Hilfsprogrammen, um aus dem produktiven Datenbestand entsprechend passende Testdaten herauszusuchen.
2. Das DiVA-Team nimmt eine beliebige Datei, die dem zu testenden Dateiformat entspricht, und modifiziert diese dann in Richtung auf die zu testende Funktionalität manuell.
3. Das DiVA-Team erzeugt eine passende Testdatei mit einer hierfür geeigneten Software (z.B. eine XML-Datei mit einem Texteditor oder eine E-Book-Datei mit einem WYSIWYG-Editor wie Sigil[[3]](#footnote-3)).

Bei diesen Strategien gibt es eine Vielzahl von Problemen, die einerseits eine effiziente Testdurchführung und andererseits auch die Testqualität beeinträchtigen können:

* Oft werden Testdaten benötigt, die eine Eigenschaft aufweisen, für die in der E-Book-Plant oder in externen Tools keine explizite Suchoption zur Verfügung steht.
* Die manuelle Modifikation von Dateien bedarf teilweise umfangreicher Kenntnisse der jeweiligen Dateispezifikation (z.B. eines XML-Schemas für ONIX oder des EPUB-Standards) - Formatfehler können nicht nur durch unzureichende Kenntnis der Spezifikation, sondern auch durch Irrtum und Tippfehler verursacht werden und die Datei für den eigentlichen Testzweck unbrauchbar machen.
* Zur Datenerzeugung verwendete Drittsoftware kann u.U. Lizenzkosten erfordern, die durch eine sporadische Nutzung nur zu Testzwecken nicht gerechtfertigt werden.
* Die Installation, Dokumentation, Anwendung verschiedener Werkzeuge für verschiedene Arten von Testdateien erfordert einen erhöhten Verwaltungs- und Kommunikationsaufwand.
* Es können bei mehrfacher Verwendung von jeweils nur leicht modifizierten Testdaten (z.B. dieselbe ISBN[[4]](#footnote-4) für unterschiedliche Testprodukte) Datenaktualisierungen in der E-Book-Plant stattfinden, die äußerst unrealistisch sind und Testergebnisse verfälschen können.

## Projektziel

Im Rahmen dieser Projektarbeit sollen zumindest die vier zuletzt genannten Probleme adressiert werden, indem eine lizenzfreie Anwendung entwickelt wird, in der zentral alle häufig benötigten Dateiformate für Testzwecke erzeugt werden können. Eine Konfiguration soll über eine grafische Benutzeroberfläche ermöglicht werden, in der sich die Anwenderinnen und Anwender des DiVA-Teams auf die jeweils relevanten Parameter konzentrieren können und darüber hinaus auf die Testdaten erzeugende Anwendung verlassen können, was die Erzeugung korrekter Dateien betrifft. So soll eine erzeugte XML-Datei stets wohlgeformt sein und eine EPUB-Datei eine Validierung mit dem epubcheck[[5]](#footnote-5) bestehen.

Darüber hinaus sollen die erzeugten Testdaten realistisch sein, also in zentralen Aspekten tatsächlichen Verlagsdaten entsprechen. Für Parameter, die sich nicht über die grafische Benutzeroberfläche des Testdatengenerators anpassen lassen, soll wo nötig oder sinnvoll über Pseudozufallsgeneratoren eine gewisse Varianz im Datenbestand erzeugt werden. Bei einigen Eigenschaften der erzeugten Daten (z.B. den Titeln oder ISBNs der Produkte) wird eine Eindeutigkeit angestrebt, so dass ein neues Testprodukt nicht ein älteres Testprodukt beim Import in die E-Book-Plant überschreibt und nicht zu ungewünschten Seiteneffekten führt.

Zusammengefasst sieht das Projektziel also die Implementierung eines Testdatengenerators vor, der folgende Eigenschaften erfüllt:

Der Testdatengenerator bietet eine teils konfigurationsbasierte, teils pseudozufällige Erzeugung von realistischen Testdaten für die E-Book-Plant. Dabei sollen die folgenden Dateitypen abgedeckt werden, die beim Testen neuer Funktionalität der E-Book-Plant häufig benötigt werden:

* Produkte: E-Books in den Formaten EPUB, PDF und Mobipocket; Hörbücher und Software in Form von Zip-Containern.
* Coverabbildungen und sonstiges Grafikmaterial in den Formaten JPG und PDF.
* Metadaten im XML-Vokabular ONIX.

Anwenderinnen und Anwender des Testdatengenerators sind die Testerinnen und Tester der E-Book-Plant, die über technisches Verständnis und umfangreiche Kenntnisse des E-Book-Markts, aber nicht notwendigerweise über Programmierkenntnisse verfügen. Die Konfiguration des Testdatengenerators sollte deshalb über eine grafische Benutzeroberfläche möglich sein.

## Übersicht über diese Projektdokumentation

In dieser Projektdokumentation wird zunächst eine kurze Übersicht über die benötigten Dateiformate gegeben. Dann erfolgt eine kurze Beschreibung der grafischen Benutzeroberfläche. Anschließend folgen einige grobe Überlegungen zur verwendeten Technik und Architektur. Anhand einer Beschreibung der Paketstruktur werden wichtige Implementierungsentschei-dungen an Beispielen erläutert.

Der Anhang enthält schließlich den Sourcecode und eine lauffähige, ausführbare JAR-Datei des Testdatengenerators auf CD-ROM.

# Beschreibung der zu erzeugenden Dateiformate

## ONIX

ONIX for Books ist ein XML-Vokabular, das vom internatiolen Verlegerverband editeur.org spezifiziert und standardisiert wird.[[6]](#footnote-6) Praxisrelevant sind die beiden Versionen ONIX 2.1 und ONIX 3.0, die beide in zwei Varianten existieren, jeweils einer mit kurzen und einer mit langen Elementnamen. Der Testdatengenerator soll beide Versionen in beiden Varianten unterstützen, also insgesamt vier verschiedene Arten von ONIX-Dateien erzeugen können.

Eine ONIX-Datei enthält Metadaten und verkaufsrelevante Informationen über Bücher und andere buchhandelsrelevante Produkte, z.B. Titel, Autoren, Verlagsangaben, Seitenzahl, Illustrationen, Verkaufsrechte und Verkaufspreise. Die E-Book-Plant unterstützt nur eine Teilmenge aller ONIX-Elemente, und zwar solche, die relevant für die Kommunikation von Daten über E-Books sind. In der Regel sind die ONIX-Dateien, die die E-Book-Plant importiert und exportiert, valide in Bezug auf die DTDs bzw. Schemas von editeur.org. Eine Validierung innerhalb des Testdatengenerators wird nicht durchgeführt, aber die erzeugten ONIX-Dateien sollen dennoch valide sein.



Abbildung 1: ONIX-Formate - Beispiel eines Knotens mit Preisinformationen

Da der Testdatengenerator vier verschiedene ONIX-Varianten handhaben muss, wurde eine Implementierung in Anlehnung an das Erbauer-Entwurfsmuster (Builder) angestrebt, bei dem einzelne Erbauer-Objekte für Teile der ONIX-Daten (z.B. Titelangaben oder Autoren) zuständig sind. Für jedes Produkt werden die Metadaten in allen vier Varianten ausgegeben, auch wenn die Testerinnen und Tester zum Import in die E-Book-Plant jeweils nur eine Variante benötigen.

## E-Book-Dateien

E-Book-Dateien im eigentlichen Sinne haben die Formate EPUB (ein IDPF-Standard[[7]](#footnote-7)), PDF (ein ISO-Standard[[8]](#footnote-8)), Mobipocket (ein propietäres Format der Firma Amazon) oder iBook (ein proprietäres Format der Firma Apple). EPUB und PDF als offene Standards lassen sich direkt oder über frei erhältliche Java-Bibliotheken erzeugen. Mobipocket-Dateien werden mit einem von Amazon zur Verfügung gestellten Tool (Kindlegen[[9]](#footnote-9)) aus EPUB- oder HTML-Dateien erzeugt. iBooks-Dateien werden in einem WYSIWYG-Editor (iBooks Author) erstellt[[10]](#footnote-10).

Eine EPUB-Datei besteht aus einem Zip-Container, in dem sich einige Meta- und Steuerdaten in XML-Formaten befinden. Die eigentlichen Buchinhalte sind per HTML-Dateien definiert, wobei Grafiken oder sonstige Medien über die in HTML üblichen Verweismechanismen eingebettet sein können. Gestaltungsinformationen sind wie in der HTML-Welt üblich per CSS festgelegt. EPUB-Dateien sind oft für die Anzeige auf kleinen Bildschirmen von Mobilgeräten optimiert (Smartphone, Tablet-Computer, E-Book-Lesegeräte mit eInk-Display), lassen sich mit entsprechender Software[[11]](#footnote-11) aber auch auf PCs anzeigen. Der Zeilenumbruch passt sich in der Regel der jeweiligen Displaygröße an, Text lässt sich unabhänig von Abbildungen skalieren.

PDF ist ein aus PostScript heraus gewachsenes, plattformunabhängiges Seitenbeschreibungsformat, bei dem die visuelle Gestaltung und insbesondere die Seitenproportionen vom Ersteller der Datei festgelegt und unveränderlich sind. PDF wird nicht nur für E-Books eingesetzt, sondern ist als allgemeines Dokumentenaustauschformat weit verbreitet. E-Books im PDF-Format sind unbequem für kleine Geräte, bieten aber Vorteile bei der Umsetzung von Fachbüchern mit hohem Grafikanteil oder bei der Anforderung, dass eine E-Book-Ausgabe zitierfähig wie ein Printbuch sein muss.

Der Testdatengenerator erzeugt diese beiden offen standardisierten Dateitypen direkt, wobei zur Erzeugung von EPUB eine XML-Bibliothek und für PDF eine PDF-Bibliothek für Java verwendet werden. Außerdem wird eine zur Verwendung in KindleGen optimierte EPUB-Datei erzeugt werden („EpubMobi“ = eine EPUB-Datei zur Erzeugung einer Mobipocket-Datei). Aus lizenzrechtlichen Gründen erfolgt keine Integration von KindleGen in den Testdatengenerator, eine Anbindung an einen von den Nutzerinnen und Nutzern selbst installierte KindleGen-Version ist hingegen möglich. Echte iBooks können gar nicht automatisiert erzeugt werden, der Testdatengenerator kreiert stattdessen Dummy-Dateien mit der Dateiendung .iBooks.

E-Book-Dateien im EPUB-Format müssen eine Validierung mit dem epub-validator des Inter­national Digital Publishing Forum ([www.idpf.org](http://www.idpf.org/)) bestehen[[12]](#footnote-12). Der Testdatengenerator erzeugt die EPUB-Dateien in der am weitesten verbreiteten Version EPUB 2 des Standards, eine Erweiterung auf das neuere Format EPUB 3 ist denkbar.

Im E-Book-Markt spielt neben den eigentlichen E-Book-Formaten auch noch die verwendete Schutzart eine große Rolle. Neben der Möglichkeit, ungeschützte Dateien zu verkaufen, werden vor allem zwei Varianten eingesetzt: Digital-Rights-Management-Systeme verschlüsseln die E-Books und Ermöglichen eine Verwendung nur legitimierten Lesegeräten bzw. -systemen. Diese Variante wird oft als „hartes DRM“ bezeichnet. Daneben werden häufig sowohl sichtbare als auch unsichtbare Wasserzeichen zum Schutz verwendet. In den Wasserzeichen können personalisierte Kundeninformationen enthalten sein, so dass bei einer illegalen Weitergabe eine Identifizierung der Person, die die derart geschützte Datei in Umlauf bringt, erfolgen kann. Die Schutzart eines E-Books spielt für den Testdatengenerator nur im Rahmen von Metadaten eine Rolle. Eine technische Anbindung an DRM-Systeme ist für die erzeugten Testdaten nicht erforderlich.

Neben E-Book-Dateien im eigentlichen Sinne verwaltet die E-Book-Plant auch elektronische Hörbücher und Softwaredownloads als Produkte. Diese beiden Produktarten werden durch Zip-Dateien repräsentiert. Es genügt hier, wenn der Testdatengenerator Dummy-Dateien nach bestimmten Benennungskonventionen in ein Zip-Archiv steckt.

Neben den eigentlichen Verkaufsprodukten verarbeitet die E-Book-Plant auch Leseproben bzw. Demoprogramme im Falle von Softwaredownloads. Diese Produktausschnitte („Extracts“) unterscheiden sich für die Zwecke des Testens nur durch eine Dateibenennungskonvention von vollwertigen Produktdateien. Der Testdatengenerator unterscheidet deshalb nicht inhaltlich zwischen einer Voll- und einer Ausschnittsversion.

## Grafik-Dateien

Das Wort „Cover“ macht eigentlich nur bei physischen Objekten Sinn, wird aber im übertragenen Sinne auch für Downloadprodukte verwendet, die auf Shop-Websites mit einem entsprechenden Produktbild präsentiert werden. Diese Produktbilder werden in der Regel im allgemein bekannten und weit verbreiteten JPEG-Format ausgetauscht.[[13]](#footnote-13) Die E-Book-Plant akzeptiert aber auch Cover-Abbildungen im PDF-Format, die sie dann beim Import in JPEG-Dateien umwandelt. Coverdateien werden unterschieden in Front Cover, Square Cover und Back Cover. Außerdem sind „Innenansichten“ oder sonstige Produktpräsentationen als „Screenshot“ bzw. 3D-Packshot möglich (alle JPEG).

# Kurzbeschreibung der grafischen Benutzeroberfläche

Nach dem Programmstart öffnet sich das Hauptfenster des Programms mit einer Buttonleiste und einer tabellarischen Ansicht von drei beispielhaften Testszenarien. Weitere Szenarien können hier angelegt, bearbeitet, gelöscht, gespeichert und geladen werden. Auch lässt sich für ein markiertes Szenario das Generieren der auszugebenden Testdateien anstoßen.

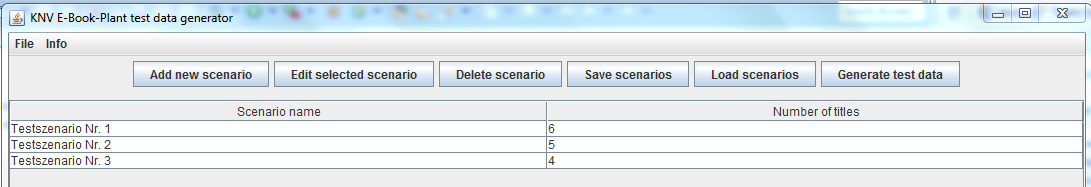


Abbildung 2: Das Hauptfenster mit den Beispielszenarien

Ein Testszenario bündelt eine beliebige Anzahl von Produkten, deren zugehörige Dateien jeweils gemeinsam erzeugt und gespeichert werden. Die Einheit eines solchen “Szenarios” ergibt sich nicht aus externen Zusammenhängen: Ein Szenario soll alle Testproduktdaten bündeln, die für die Abnahmetests einer umgrenzten neuen Funktionalität der E-Book-Plant benötigt werden.

Die Bearbeitungsansicht eines neuen oder bestehenden Szenarios besteht wieder im Wesentlichen aus einer Buttonleiste und einer Tabelle. In der Tabelle ist je Testprodukt (“Title”) eine Zeile vorgesehen. Titel lassen sich über die Buttons neu hinzufügen oder entfernen. Beim Hinzufügen öffnet sich ein drittes, modales Fenster, in dem Detailangaben zu bestimmten Testprodukten konfiguriert werden. Diese Angaben beziehen sich in der Regel auf Daten des Onixformats oder Dateitypen der E-Book-Plant. Den Anwenderinnen und Anwendern der E-Book-Plant sind diese Daten und deren Verwendung jeweils bekannt und somit nicht weiter erläuterungsbedürftig. Für Außenstehende hingegen mögen die Auswahlmöglichkeiten etwas esoterisch erscheinen.

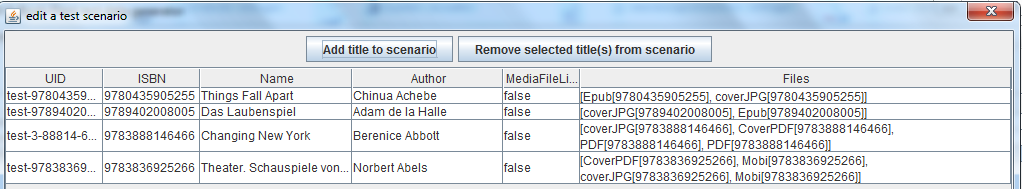


Abbildung 3: Bearbeitungsansicht eines Testszenarios

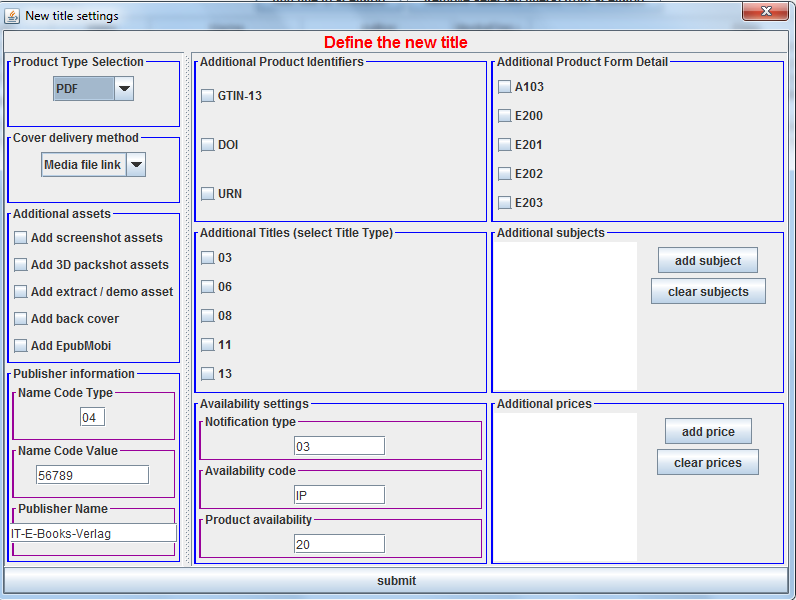


Abbildung 4: Detailkonfiguration eines Produkts

Beim Speichern oder Laden eines Szenarios wird die Datei über FileChooser-Dialoge des Java-Swing-Frameworks ausgewählt; eine Speicherung erfolgt mit der Endung .ebp und mit Hilfe der in Java eingebauten Serialisierungstechnologien.[[14]](#footnote-14)

Auch vor dem Erzeugen der zu einem Szenario gehörenden Testdateien öffnet sich ein FileChooser-Dialog, um das Zielverzeichnis der zu generierenden Dateien festzulegen.

# Architekturübersicht

Zur Strukturierung des Quellcodes wird eine Einteilung in Pakete vorgenommen, die sich grob an verbreiteten Entwurfsmustern orientiert. Für einige Funktionen werden externe Bibliotheken eingebunden. Die Verwendung von Java-Standardbibliotheken wird dabei vorausgesetzt und nicht weiter explizit motiviert. Dies gilt auch für die Wahl von Swing als GUI-Framework, welches derzeit von JavaFX peu à peu abgelöst wird. Da an die grafische Benutzeroberfläche keine besonderen Anforderungen zu stellen sind, war eine Umsetzung mit beiden GUI-Frameworks denkbar und die Wahl fiel aus nicht-funktionalen Gründen auf dasjenige, mit dem der Programmierer bereits vertrauter war.

Im Folgenden wird nun eine Überblick über die Grobstrukturierung gegeben. Die beiden grundlegenden Modellelemente wurden bereits oben bei der Beschreibung der grafischen Benutzeroberfläche erwähnt: Testszenario und Produkt. Diese werden durch die Klassen TestScenario und Title im Paket testdatagen.model repräsentiert. Auch für jede Art der zu erzeugenden und auszugebenden Dateien gibt es eine eigene Klasse. Deren Definitionen sind im Paket testdatagen.model.files zusammengefasst.

Da sich ein Großteil der Konfigurationen des Testdatengenerators in XML-Elementen der zu erzeugenden ONIX-Dateien niederschlägt, wurde hierfür ein eigenes Paket definiert: testdatagen.onixbuilder.

Die Klassen zur Verwaltung der grafischen Benutzeroberfläche wurden auf die beiden Pakete testdatagen.gui und testdatagen.gui.listeners verteilt. Schließlich gibt es noch kleinere Pakete für statische Hilfsmethoden (testdatagen.utilities) und die Verwaltung von statischen Konfigurationsdaten (testdatagen.config). Das Hauptprogramm und einige Klassen, die Arbeitsthreads repräsentieren, befinden sich im „Hauptpaket“ testdatagen.

Das Programm wurde mit Hilfe der Java-IDE Eclipse entwickelt und dort zu ausführbaren JAR-Dateien kompiliert. Dem DiVA-Team wird es auf einem Netzlaufwerk zur Verfügung gestellt. Mehrere Instanzen des Programms können parallel betrieben werden, wobei ein synchronisierter Zugriff auf eine Datei erfolgt, in der die zuletzt verwendete ISBN und die zuletzt verwendete Titelnummer gespeichert sind. Hierdurch wird auch bei Verwendung mehrerer paralleler Instanzen sichergestellt, dass die verwendeten ISBNs nur einmal vergeben werden.

## Die Pakete des Testdatengenerator-Quellcodes

### testdatagen.model

Im Modellpaket finden sich zunächst die beiden zentralen Datenklassen Title und TestScenario. Die Title-Klasse kapselt alle Informationen zu einem einzelnen Testprodukt, wohingegen die TestScenario-Klasse eine Zusammenstellung von beliebig vielen Title-Objekten zu einem größeren Testszenario darstellt.

Der Testdatengenerator wurde nicht vollständig nach dem Model-View-Controller-Muster (MVC) aufgebaut. Ein Title-Objekt wird beim Hinzufügen zu einem Testszenario erzeugt, kann aber danach durch die Nutzerin des Testdatengenerators nicht mehr modifiziert werden. Aus diesem Grund müssen Änderungen am Title-Modell durch die View nicht überwacht werden und das Title-Modell bietet aus diesem Grund auch keine entsprechenden Methoden an, sprich: das Modell ist nicht Observable.

Zwei Klassen, die jedoch Änderungen während des Programmablaufs unterliegen und somit in der View aktualisiert werden müssen, stellen Modelle im Sinne des MVC-Musters dar: Im TitleTableModel werden alle zu einem Testszenario gehörigen Title-Objekte verwaltet und im ScenarioTableModel werden alle zu einem Programmlauf gehörenden Testszenarien aufgelistet. Diese beiden Klassen sind vom AbstractTableModel des Swing-Frameworks abgeleitet und integrieren sich damit auf natürliche Weise in die MVC-Struktur, die das Swing-Framework für Tabellen zur Verfügung stellt: Das View-Element JTable registriert sich als Listener bei einem TableModel und wird bei Änderungen am TableModel informiert, so dass die Anzeige aktualisiert werden kann. Zu diesem Zweck rufen die Objekte der Klassen TitleTableModel und ScenarioTableModel die Methode fireModelChanged bzw. fireTableRowsInserted oder fireTableRowsDeleted auf, wenn Objekte zum Modell hinzugefügt oder entfernt werden.

Die TestScenario-Klasse delegiert dabei die Verwaltung der zum Szenario gehörenden Titel vollständig an das TitleTableModel und hält selbst keine direkten Referenzen auf die Title-Objekte (vgl. dazu das UML-Klassendiagramm in der folgenden Abbildung 5).

Über diese vier zentralen Klassen hinausgehend enthält das Model-Paket noch einige Hilfsklassen, die einzelne Aspekte von Titeln wie Preise, Produktart und Schlagwort in einer Art und Weise kapseln, die mit der Modellierung dieser Informationen in ONIX oder in der E-Book-Plant übereinstimmt.

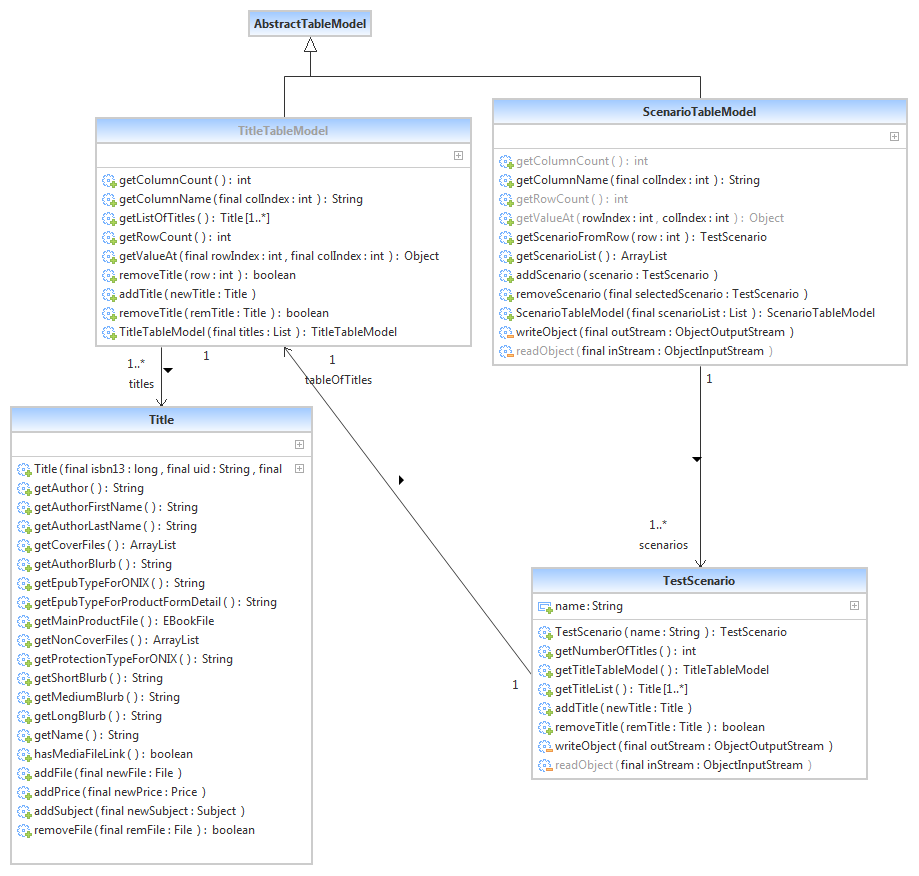


Abbildung 5: Zentrale Modellklassen

### testdatagen.model.files

Während die unter 4.1.1 genannten Modellklassen Konfigurationsdaten verwalten, repräsentieren die Klassen im Paket testdatagen.model.files die zu erzeugenden Testdateien. Sie sind alle von einer abstrakten Klasse File abgeleitet, welche den abgeleiteten Klassen die Implementierung der abstrakten Methoden buildFileName() und generate(File destFolder) vorschreibt. Dabei ist buildFileName() für die Festlegung von Dateinamenskonventionen der jeweiligen Dateiart zuständig und generate() erzeugt die eigentlichen Dateien in einem anzugebenden Zielverzeichnis.

Um weitere Spezifika bestimmter Dateisorten abbilden zu können, werden zwei weitere abstrakte Klassen verwendet: GraphicFile für Abbildungsdateien, EBookFile für die unter 2.2 vorgestellten E-Book-Dateien. Da Metadaten stets im Dateiformat ONIX ausgegeben werden, war hier keine weitere Zwischenstufe mehr nötig und ONIXFile ist somit nicht abstrakt definiert. Einen Überblick über dieses Paket gibt das UML-Diagramm in Abbildung 6.

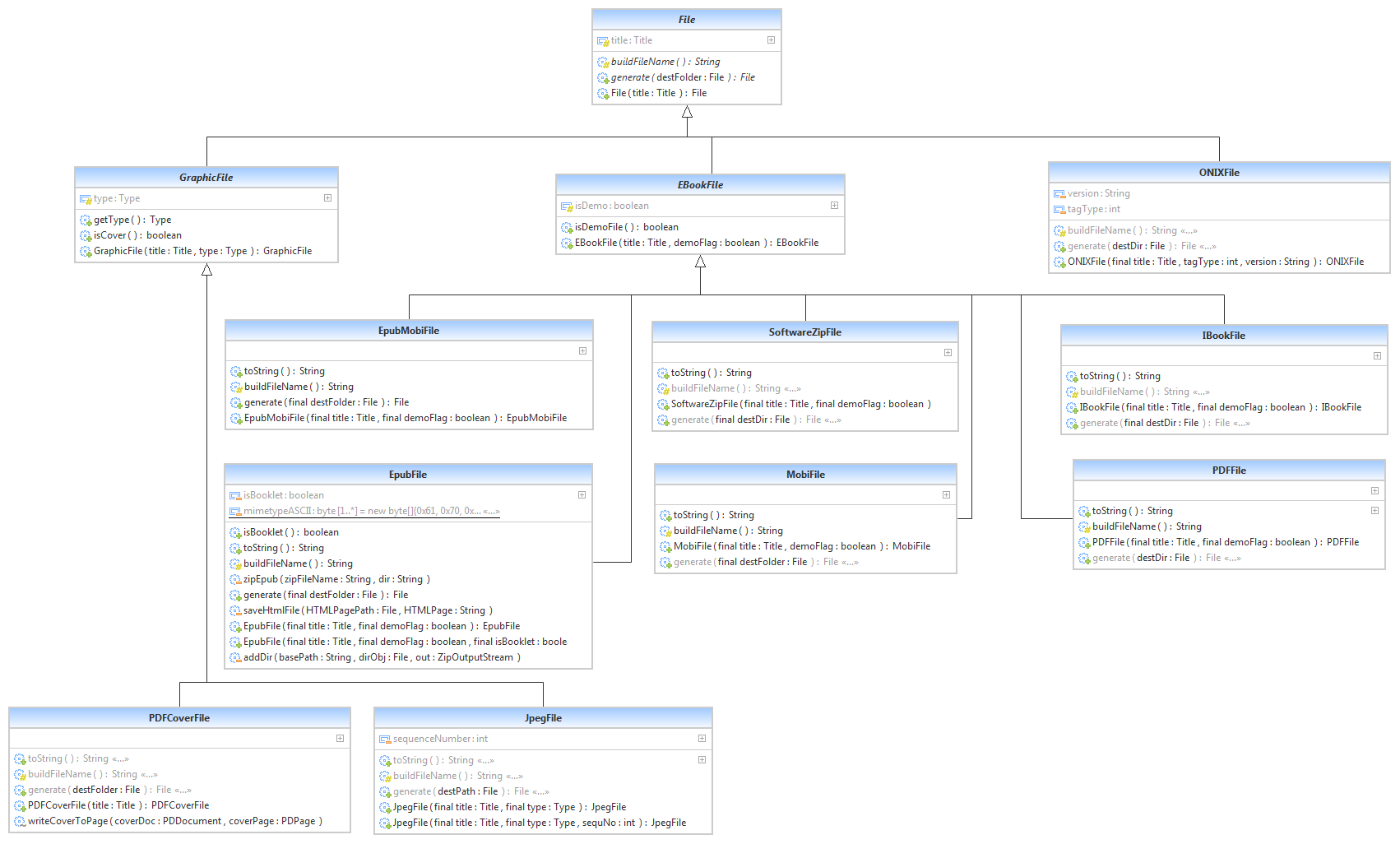


Abbildung 6: Das Dateimodell

Die Title-Objekte verwalten eine Liste der jeweils für das Produkt benötigten Dateien, so dass sich der in Abbildung 7 gezeigte einfache Zusammenhang zwischen den beiden Paketen testdatagen.model und testdatagen.model.files ergibt.

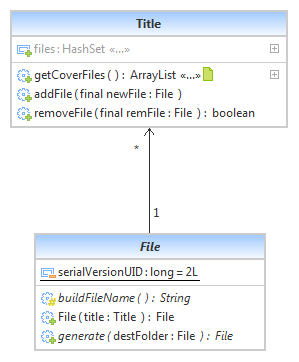


Abbildung 7: Title und Files

### testdatagen.onixbuilder

Da für jedes Produkt äquivalente Onix-Dateien in vier verschiedenen Varianten erzeugt werden müssen, bietet sich eine Modellierung nach dem Erbauer-Entwurfsmuster an. Die Metadaten einer Onix-Datei werden nach semantisch-syntaktischen Gesichtspunkten auf verschiedene Klassen verteilt, die alle von der abstrakten Klasse OnixPartsBuilder abgeleitet sind (siehe Abbildung 0). Eine konkrete OnixPartsBuilder-Klasse lässt sich über ein Argumente-Dictionary (Map<key, value>) konfigurieren, enthält ein zweidimensionales Array zur Definition der XML-Elementnamen, liefert eine SequenceNumber zur Reihenfolgensortierung der Onix-Teile in der zu erzeugenden Onixdatei und bietet die zentrale Funktionalität, nämlich die Instanziierung der gewünschten Onix-XML-Elemente über die build()-Methode an, welcher als Argumente die Onixversion (2.1 bzw. 3.0) und die Tag-Art („short tags“ vs. „reference tags“) übergeben werden.

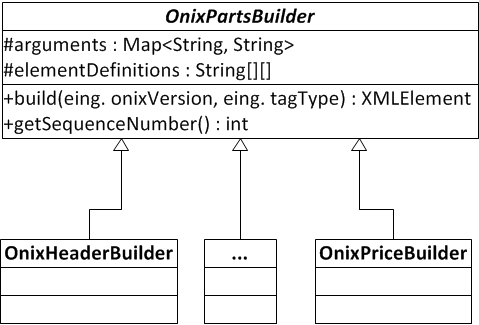


Abbildung 8: Die OnixPartsBuilder-Klasse

Eine Onixdatei enthält in der Regel verpflichtende und optionale Elemente. Um einen flexiblen Bauplan für die konkreten Onixdateien zu ermöglichen, wird die Konfiguration über ein OnixPartsDirector-Objekt gesteuert. Ein OnixPartsDirector-Objekt verwaltet die nötigen Abschnitte bzw. Elemente in einer requiredElements-Liste. Der Konstruktor der OnixPartsDirector-Klasse sorgt dafür, dass zumindest alle verpflichtenden Onixteile zu den requiredElements hinzugefügt werden. Um weitere, optionale Elemente hinzuzufügen, bietet die OnixPartsDirector-Klasse AddXXX()- oder ReplaceXXX()-Methoden an. Ist eine Onixdatei via OnixPartsDirector-Objekt fertig konfiguriert, lässt sich der komplette XML-Baum einer Onixdatei durch Aufruf der Methoden buildOnix2() bzw. buildOnix3() erzeugen. Diese Erbauer-Methoden bringen zunächst alle requiredElements in die schemakonforme Reihenfolge und delegieren die Erzeugung der XML-Elemente dann an die build()-Methoden der konkreten OnixPartsBuilder-Kindobjekte. Das Zusammenspiel zeigt Abbildung 9.

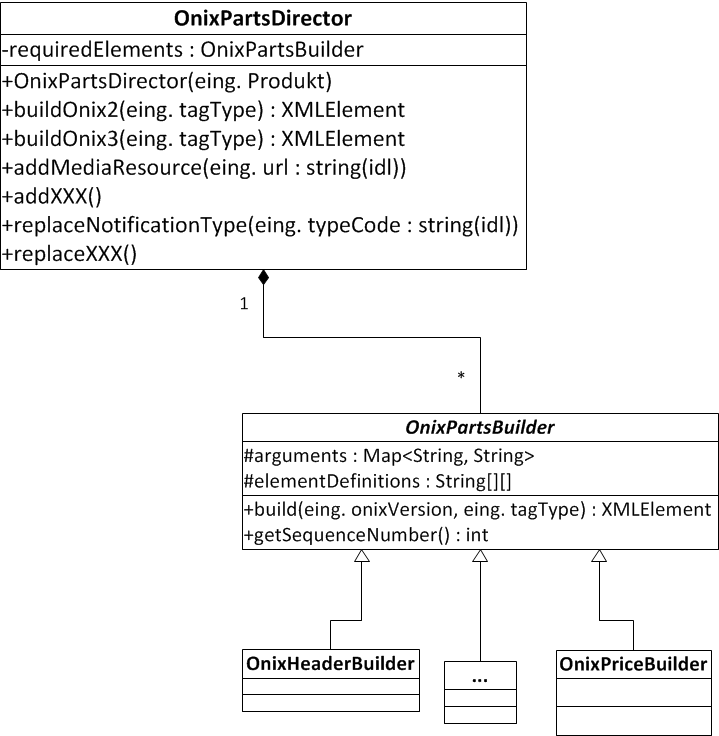


Abbildung 9: OnixPartsDirector

### testdatagen.gui

Wie unter [3.1.1.](#_toc472) erläutert, ist nur ein kleiner Teil der Modellklassen „observable“, so dass nur sehr punktuell von einem Model-View-Controller-Modell gesprochen werden kann. Aus diesem Grund wurde das Paket zur grafischen Darstellung der Konfigurationen nicht „view“, sondern „gui“ genannt, um keine falschen Vorstellungen zu wecken. Das Paket enthält im Wesentlichen Formulardialoge, die bei der Definition neuer Datensätze für Testprodukte oder für ganze Testszenarien angezeigt werden. Hierfür wurden jeweils Komponenten aus dem Java-Swing-Framework verwendet bzw. wo nötig eigene Klassen davon abgeleitet.

Da sowohl die Szenarienübersicht als auch die Produktübersicht innerhalb von Szenarien durch Tabellen realisiert wird, wurde ein eigener TableCellLineWrapRenderer definiert, der für Zeilenumbruch und die optimale Höhe und Breite der Tabellenspalten und -zeilen sorgt.

### testdatagen.gui.listeners

Aus demselben Grund, aus dem das gui-Paket nicht „model“ heißt, implementieren auch die Listener keine „controller“. Dennoch erschien aufgrund der relativ großen Zahl an nötigen Listenern für Elemente der grafischen Benutzeroberfläche eine Strukturierung durch ein eigenes Paket angebracht. Die Objekte der einzelnen Listenerklassen dienen, wie im Swing-Framework üblich, zur Reaktion auf Button- oder Menüklicks und leiten entsprechende Aktionen ein. In einigen Fällen definieren die Listener wiederum eigene, kurze Dialogfenster (z.B. der AddPricesToTitleListener als Hilfsklasse PricesDialog oder MenuAboutInfoListener über ein lokal in der actionPerformed()-Methode generiertes JFrame). Maßgabe für die Entscheidung, ob die Listenerklassen eigene grafische Komponenten definieren und anzeigen oder ob eine Komponente eine Klasse im Paket testdatagen.gui darstellt, war jeweils die Komplexität des grafischen Containers: Je größer die Anzahl der Grafikkomponenten, desto eher eine Modellierung als eigene Klasse in testdatagen.gui.

### Restliche Pakete

Im „Haupt“-Paket testdatagen findet sich die Klasse TestDataGeneratorMain mit der Main-Methode, die den Einstiegspunkt in das Programm bildet und das Hauptfenster des grafischen Programms definiert und zur Anzeige bringt. Außerdem befinden sich in diesem Paket die Klassen, welche die „Erzeugungslogik“ des Programms repräsentieren. Dazu gehören Fabrikmethoden für die Erzeugung von Objekten, welche bestimmte Ausgabedateien definieren (EbookFileFactory und GraphicFileFactory mit dem zugehörigen Interface GraphicFileFactory) und Klassen für die nebenläufige Abarbeitung bestimmter Aufgaben: GeneratorThread nimmt eine Liste von Produkten entgegen, für die alle zugehörigen Ausgabedateien erzeugt werden. Außerdem errechnet die von SwingWorker abgeleitete Klasse einen Fortschrittswert für die Fortschrittsanzeige im Hauptfenster des Programms.

Der DropboxUploaderThread lädt Coverdateien in einen fest vorgegebenen Dropbox-Account hoch und erzeugt Freigabe-URLs für diese Dateien. Diese URLs werden wiederum in die Onix-Meldungen der zugehörigen Produkte eingefügt, so dass über den Onix-Mechanismus des <MediaLink>-Elements URLs angegeben werden können, hinter denen tatsächlich im Internet erreichbare Coverdateien liegen.

In dem Paket testdatagen.config sind Konfigurationsdaten zusammengefasst. Diese sind etwas unelegant hart im Java-Programm codiert, doch wurde bei der Implementierung der Klasse ConfigurationRegistry darauf geachtet, die Konfigurationsinformationen leicht in eine oder mehrere externe Dateien auslagern zu können. In diesem Falle müsste nur der Konstruktor geändert werden, der dann die privaten Kollektionen mit Konfigurationsdaten nicht mehr direkt befüllen würde, sondern die entsprechenden Daten dann aus externen Dateien laden könnte. Aktuell bestand bei den Nutzerinnen und Nutzern des Testdatengenerators keine Anforderung, die hier versammelten Konfigurationsdaten selbst vor dem Programmstart modifizieren zu können, so dass hier bislang auf die vereinfachte Variante gesetzt werden konnte. Die ebenfalls in dem Paketordner liegende Datei chapterSentences.xml versammelt Sätze, aus denen in den erzeugten E-Book-Dateien „Zufallsromane” erzeugt werden.

Das Paket testdatagen.templates bietet in erster Linie einen Mechanismus, Textmuster mit Variablen vorgeben zu können und die Variablen dann später mit zufällig gewählten Texten zu befüllen. Die Textmustertemplates sind in der oben besprochenen ConfigurationRegistry definiert und enthalten fixen Text und Variablen in der Form {$variablenname}. Das Verhalten eines speziellen Templates ist in der abstrakten Basisklasse TextTemplate definiert. Jede Subklasse muss eine Methode fillWithText() anbieten, die die Ersetzung der Variablen in den Templates vornimmt. Dazu steht die Hilfsmethode replaceVars(...) zur Verfügung, die den Templatetext rekursiv durchläuft und für die Ersetzung der Variablen durch Werte sorgt. Der hier geschilderte Template-Mechanismus wird im Testdatengenerator sowohl zur Erzeugung bestimmter Werbetexte in Onix-Meldungen verwendet (AuthorBlurbTemplate,TitleBlurbTemplate) als auch für Dummytexte innerhalb von E-Book-Dateien (EbookChapterTemplate, EPUBCoverPageTemplate, EPUBTitle­Page­Template).

Das Paket testdatagen.utilities schließlich enthält statische Hilfsmethoden, für die aus inhaltlichen, semantischen Gründen keine passende Klasse zur Verfügung stand oder die programmglobale Aufgaben erledigen. Eine solche stellt z.B. getNextISBN() in der Klasse ISBNUtils dar. Jede neu verwendete ISBN wird gespeichert, so dass die Eindeutigkeit der ISBNs durch stetiges Hochzählen gewährleistet wird und persistent auch über Programmneustarts hinweg funktioniert.

## Verwendete externe Bibliotheken

Zum Erzeugen bestimmter Ausgabeformate (PDF, XML) werden Bibliotheken verwendet, die eine objektorientierte Modellierung und Erzeugung dieser Dateiformate erlauben. Wo möglich, sollten Open-Source-Bibliotheken eingesetzt werden. Im Falle von PDF wurde aus diesem Grund Apache PDFBox[[15]](#footnote-15) gewählt, im Falle von XML (für ONIX- und EPUB-Dateien) wurde XOM[[16]](#footnote-16) eingesetzt. Aus den Apache-Bibliotheken wurden zudem einige Funktionen der Apache Commons IO-Bibliothek hinzugezogen, um einen robusten, betriebssystem­übergreifenden Dateizugriff zu realisieren.

Zum Upload von Grafikdateien zum Onlinespeicherdienst Dropbox wurde schließlich noch die proprietäre Bibliothek Dropbox-Core-SDK eingebunden, welche wiederum Jackson-Core zur Verarbeitung von Daten im JSON-Format erfordert.

## Codestil-Konventionen

Innerhalb einer Klasse sind zunächst alle öffentlichen und privaten Instanzvariablen in der Reihenfolge öffentlich -> privat angegeben. Es folgt stets der Konstruktor (sofern vorhanden) und anschließend zunächst alle öffentlichen, dann alle geschützten (protected), dann alle privaten Methoden. Innerhalb einer Methodenkategorie erfolgte eine alfabetische Sortierung nach dem Methodennamen.[[17]](#footnote-17)

Alle öffentlichen Methoden werden mit Kommentaren im JavaDoc-Stil eingeleitet. Private Methoden sind je nach Bedarf kommentiert. Alle Kommentare wurden in Englisch verfasst, denn der Quellcode wurde in ein (derzeit jedoch nicht-öffentliches) Github-Repositorium eingestellt, wo er potentiell „mit der Welt geteilt“ werden kann.[[18]](#footnote-18)

# Zusammenfassung und Ausblick

Die angepeilten Ziele, dem DiVA-Team eine lizenzfreie, konfigurationsgesteuerte Möglichkeit zum Erzeugen von Testdaten für die E-Book-Plant, konnten mit dem hier vorgestellten Testdatengenerator verwirklicht werden. Während der Umsetzungsphase dieser Projektarbeit sind dem DiVA-Team Betaversionen des Testdatengenerators zur Verfügung gestellt und in der praktischen Arbeit erprobt worden. Dabei war das Feedback des DiVA-Teams positiv, die Verwendung des Testdatengenerators wurde als Vorteil gegenüber der vorherigen Situation (Suche bzw. manuelle Erzeugung passender Testdateien) empfunden.

Die von den Betaversionen erzeugten Testdaten wurden zum Testen von Programm­änderungen an der E-Book-Plant erfolgreich und gewinnbringend eingesetzt. Die Testdaten erfüllten die Forderungen an die Korrektheit der Dateiformate und an die Konformität mit besonderen Anforderungen der E-Book-Plant. Auch erwies es sich in vielen Fällen tatsächlich als zeitsparender, Testdaten mit dem Testdatengenerator zu erzeugen - verglichen mit den anderen Möglichkeiten wie der Suche nach geeigneten Testdaten oder deren manueller Erstellung.

Dennoch ergab sich im Laufe der Planung und Umsetzung eine ganze Reihe von Desiderata, die sich noch realisieren ließen. Diese beziehen sich einerseits auf die Unterstützung weiterer Dateiformate oder Konfigurationsmöglichkeiten, andererseits auf Verbesserungen in Entwurf, Architektur und Programmcodestruktur.

## Ausblick 1: Ausbau fachlicher Funktionalität

Eine vorerst noch nicht abgedeckte Besonderheit bei den Dateiformaten, die die E-Book-Plant verarbeitet, stellt EPUB3 dar. Im E-Book-Markt werden beide Formate parallel eingesetzt, wobei aus Gründen der Unterstützung älterer Lesegeräte die Verwendung der Version 2 der EPUB-Spezifikation noch dominiert. Dies ist aus diesem Grund auch das Format, welches der Testdatengenerator für EPUB-Produkte generiert. Nichtsdestotrotz sind Testszenarien denkbar, in denen dezidiert EPUB3-Dateien eingesetzt werden müssen. Sie sollten deshalb eine eigene Konfigurationsoption erhalten und spezifikationskonform umgesetzt werden. Für eine Basisversion von EPUB3-Dateien würde es genügen, die Versionsangabe entsprechend zu setzen und ein HTML-Inhaltsverzeichnis einzufügen (EPUB 2 enthält das Inhaltsverzeichnis im NCX-Format). Diese Anpassung würde nur geringfügige Anpassungen im bestehenden Code erfordern. Darüber hinaus könnte eine spezielle Art von EPUB3-E-Books interessant sein: die sogenannten Fixed-Layout-EPUBs. Diese werden häufig bei grafiklastigen Inhalten eingesetzt (z.B. Kinderbilderbüchern und Comics) und stellen für bestimmte E-Book-Plant-Prozesse besondere Herausforderungen dar. Der Aufwand für die Erweiterung des Testdatengenerators um Fixed-Layout-EPUB3-Dateien wäre etwas größer, da Anpassungen nicht nur in den Metadaten der Datei, sondern auch im CSS erforderlich wären: Es müssen bei dieser Formatvariante sogenannte Viewports für die geplante Bildschirmgröße bzw. die geplanten Anzeigeproportionen hinterlegt werden.

Ein weiteres größeres Desiderat im Bereich der fachlichen Funktionalität bleibt die derzeit noch relativ geringe Zahl an Konfigurationsmöglichkeiten zu Produktdaten, die sich dann in den ausgegebenen ONIX-Dateien widerspiegeln. Ein Vervielfachung der derzeit angebotenen Konfigurationsoptionen ist nicht unwahrscheinlich. Im Rahmen des verwendeten Erbauer-Entwurfsmusters lassen sich weitere Erzeugungsregeln für die ONIX-Dateien leicht hinzufügen. Jedoch drohen bei einer solchen Erweiterung die Klassen TitleDialog und AddTitleToScenarioListener unübersichtlich zu werden. Eine weitere Differenzierung und Binnenstrukturierung, sprich: ein Refactoring, werden hier bei einer entsprechenden Weiterentwicklung wohl schnell erforderlich.

## Ausblick 2: Verbesserung der Codestruktur und der Entwicklungs-Flexibilität

Wie in Kapitel 4.1.6 erwähnt, befinden sich Konfigurationsdaten, die beim Programmstart geladen werden, in hart kodierter Form in der Klasse ConfigurationRegistry. Um eine höhere Flexibilität (Anpassbarkeit der globalen Konfigurationsdaten durch Anwenderinnen und Anwender, nicht nur durch den Entwickler des Programms) und eine bessere Strukturierung (z.B. Verteilung von lokalisierten Konfigurationsdaten auf sprachspezifische Dateien) zu erreichen, sollte hier ein Refactoring stattfinden. An den öffentlichen Methoden der Klasse ConfigurationRegistry sind dabei keine Änderungen notwendig. Jedoch sollte der Konstruktor der Klasse die entsprechenden Daten aus Dateien einlesen und dann während des Programmlaufs vorhalten. Eine einfache Möglichkeit, dies zu realisieren, wäre die Verwendung der Properties-Klasse aus der Java-Systembibliothek java.util.

Je größer und komplexer das Programm wird, desto aufwändiger wird voraussichtlich auch das Debugging werden. Aus diesem Grunde wäre es sehr sinnvoll, Prozess- und Statusausgaben in eine Logdatei zu schreiben. Neben diesen Log-Informationen, die in erster Linie für den Entwickler interessant sind, könnte auch ein spezielles Log für Anwenderinnen und Anwender eingerichtet werden, das z.B. die Informationen des KindleGen-Tools, die dieses während der Erzeugung von Mobidateien aus E-Books im EPUB-Format auf der Kommandozeile ausgibt, in einer Datei fixiert.

In der Konfiguration ist bereits bei einigen Methoden ein Locale-Parameter eingebaut, der für eine Anpassung der Datei an verschiedene Sprach- oder Landesumgebungen verwendet werden kann. Derzeit ist der Testdatengenerator jedoch ausschließlich für die Verwendung mit deutschen Ausgabetexten (in E-Books und ONIX-Dateien) vorbereitet, während die gesamte Menüführung der grafischen Benutzeroberfläche in Englisch ausgeführt ist. Dies sieht inkonsequent aus, entspricht aber dem Usus in der E-Book-Plant. Dennoch wäre eine Erweiterung natürlich wünschenswert, die für eine volle Lokalisierung sowohl der GUI als auch der erzeugten Ausgabedaten sorgt. Beschriftungen und Buttons würden dann nicht mehr hart im Quellcode hinterlegt, sondern aus Konfigurationsdateien für die jeweils gerade eingestellte Sprach- bzw. Landeseinstellung geladen. Eine weitere unterstützte Sprache ließe sich so leicht und ohne größere Anpassungen im Programmcode einfach durch Hinzufügung einer neuen Sprachdatei realisieren.

Und schließlich besteht noch das Problem der unflexiblen Serialisierung, die gespeicherte Testszenarien aus früheren Versionen des Testdatengenerators unlesbar werden lässt - unter der nicht unwahrscheinlichen Voraussetzung, dass von einer Version zur nächsten eine Änderungen an einer der zur Speicherung verwendeten Klassen modifiziert wurde. Dies könnte durch den Entwurf und die Implementierung eigener Serialisierungsmethoden und -konstruktoren, sowie evtl. eigener Speicherformate verbessert werden.

Index

A

Abnahmetests 2

C

Cover 6, 14

D

Debugging 17

digitale Verlagsauslieferung 1

Digital-Rights-Management 6

DiVA-Team 1

Dropbox 14, 15

E

E-Book-Plant 1

EPUB 5, 16

EPUB3 16

epubcheck 3

EpubMobi 5

Erbauer 5, 12, 17

Erzeugungsregeln 17

G

Github 16

grafische Benutzeroberfläche 6, 9, 17

H

Hörbücher 6

I

iBooks 5

ISBN 9, 15

J

JPEG 6

K

Kindlegen 5

KindleGen 5

Koch, Neff & Oetinger Verlagsauslieferung GmbH 1

Kommentare 16

Konfiguration 14, 17

L

Leseproben 6

Listener 10, 14

Lokalisierung 17

M

Modell 9

Model-View-Controller 9, 14

O

Observable 14

Onix 7, 12, 14

ONIX 9, 10, 11

ONIX for Books 4

P

PDF 5, 6

S

Schutzart 6

Serialisierung 8

Softwaredownloads 6

Swing 9, 14

T

Template 15

Testszenario 7

Textmuster 15

W

Wasserzeichen 6

Glossar

|  |  |
| --- | --- |
| DiVA | Digitale Verlagsauslieferung. |
| DiVA-Team | Mitarbeiterinnen und Mitarbeiter in der digitalen Verlagsausliefernug der Koch, Neff und Oetinger Verlagsauslieferung GmbH. |
| ISBN | International Standard Book Number; ursprünglich eine zehnstellige, aber durch Anpassung an den EAN-Standard (European Article Number) bzw. an die GTIN (Global Trade Item Number) inzwischen dreizehnstellige Nummer zur eindeutigen Identifizierung von selbständig erscheinenden Verlagsprodukten. |
| KNO-VA | Koch, Neff und Oetinger Verlagsausliefernug GmbH. |
| ONIX | Ein von der internationalen Verlegervereinigung editeur.org veröffentlichter Standard zum Austausch von Metadaten über Verlagsprodukte. |

Anhang: Übersicht über die Inhalte der beiliegenden CD-ROM

Erklärung der Kandidatin / des Kandidaten

* Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.
* Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist im Kapitel „Verantwortliche“ zu Beginn der Dokumentation aufgeführt.

Diesen Teil habe ich selbstständig verfasst und keine anderen als die angegebenen   
 Quellen und Hilfsmittel verwendet.

Name der Mitverfasser: ....................................................................................................  
 ...........................................................................................................................................

|  |  |  |
| --- | --- | --- |
| Datum |  | Unterschrift der Kandidatin / des Kandidaten |

1. Zum Zeitpunkt des Verfassens dieser Arbeit handelt es sich bei dem konkreten Programmiererteam tatsächlich ausschließlich um Männer, weshalb hier aus empirischen Gründen auf eine geschlechtsneutrale Benennung verzichtet wird. Dies soll natürlich nicht ausschließen, dass zukünftig auch Programmiererinnen an der an der Entwicklung der E-Book-Plant mitwirken können. [↑](#footnote-ref-1)
2. In Bezug auf die in dieser Projektarbeit behandelte Testdatenerzeugung gibt es keinen Unterschied zwischen Abnahme- und Regressionstests, so dass hier und künftig zusammenfassend nur von Anwenderinnentests die Rede ist. [↑](#footnote-ref-2)
3. https://sigil-ebook.com/. [↑](#footnote-ref-3)
4. Internationale Standard-Buchnummer. [↑](#footnote-ref-4)
5. Epubcheck ist ein spezielles Validierungstool, um die Konformität einer EPUB-Datei in den Formaten EPUB 2 und EPUB 3 mit den Standards von editeur.org zu überprüfen. Siehe <https://github.com/idpf/epubcheck> (Zugriff am 12.03.2016). [↑](#footnote-ref-5)
6. http://www.editeur.org/11/Books, Zugriff am 6.2.2016. [↑](#footnote-ref-6)
7. <http://idpf.org/epub>, Zugriff am 6.2.2016. [↑](#footnote-ref-7)
8. <http://www.iso.org/iso/catalogue_detail.htm?csnumber=51502> ist die offizielle Seite für den Standard. Eine kostenlose Version gibt es bei Adobe, der Firma, die das Format „erfunden“ hat: <http://www.adobe.com/devnet/pdf/pdf_reference.html>. Zugriffe jeweils am 6.2.2016. [↑](#footnote-ref-8)
9. <https://www.amazon.com/gp/feature.html?docId=1000765211>, Zugriff am 6.2.2016. [↑](#footnote-ref-9)
10. <http://www.apple.com/ibooks-author/>, Zugriff am 6.2.2016. [↑](#footnote-ref-10)
11. Z.B. Adobe Digital Editions (<https://www.adobe.com/de/solutions/ebook/digital-editions/download.html>, Zugriff am 02.03.2016), Azardi Reader (<http://azardi.infogridpacific.com/>, Zugriff am 02.03.2016) oder Calibre (<https://calibre-ebook.com/>, Zugriff am 02.03.2016). [↑](#footnote-ref-11)
12. <http://validator.idpf.org/>, Zugriff am 6.2.2016.0 [↑](#footnote-ref-12)
13. <http://jpeg.org/index.html>, Zugriff am 6.2.2016. [↑](#footnote-ref-13)
14. Die eingebaute Java-Serialisierung bietet eine generische Möglichkeit, Java-Objekte zum Speichern im Dateisystem oder zum Versand über Netzwerkschnittstellen ohne eigenen Programmieraufwand zu serialisieren. Dies wird jedoch mit dem Nachteil erkauft, dass Änderungen des internen Objektmodells von einer Programm­version zur nächsten zu Inkompatibilität der gespeicherten, serialisierten Dateien führen. Eine Erweiterung der Serialisierung um Deserialisierungskonstruktoren kann dieses Problem beheben, war aber im Rahmen dieses Projekts nicht umsetzbar. [↑](#footnote-ref-14)
15. https://pdfbox.apache.org/. [↑](#footnote-ref-15)
16. http://www.cafeconleche.org/XOM/. [↑](#footnote-ref-16)
17. Hierbei wurde kein inhaltsleerer Formalismus betrieben. Wo sinnvoll, etwa bei der main()-Methode in der TestDataGeneratorMain-Klasse, wurde aus sachlichen Gründen von der Sortierung abgewichen. [↑](#footnote-ref-17)
18. Der URL des Github-Repositoriums lautet <https://github.com/sermo-de-arboribus/EBP-testdata-generator/>. Da der Code nicht öffentlich ist, benötigt man ein kostenloses Github-Konto sowie eine Freischaltung durch den Repository-Administrator. Bei Interesse an einer solchen Freischaltung senden Sie bitte eine Anfrage an [sermo\_de\_arboribus@seznam.cz](mailto:sermo_de_arboribus@seznam.cz) oder weberk@hochschule-trier.de. [↑](#footnote-ref-18)