aus

Oliver J. Bott, Peter Fricke, Uta Priss, Michael Striewe (Hrsg.)

# Automatisierte Bewertung in der Programmierausbildung

Digitale Medien in der Hochschullehre Band 6
2017, 420 Seiten, br., 42,90 €, ISBN 978-3-8309-3606-0


Waxmann Verlag GmbH
www.waxmann.com info@waxmann.com

# 24 Ein XML-Austauschformat für Programmieraufgaben

**Sven Strickroth, Oliver Müller und Uta Priss**

***Zusammenfassung***
*In diesem Kapitel wird die aktuelle Version 1.1 des ProFormA-Austauschformats beschrieben. Dabei wird insbesondere auf Änderungen zu vorhergehenden Versionen und aktuelle Erfahrungen aus vierjähriger Entwicklung und Einsatz eingegangen. Dieses Kapitel basiert auf den Publikationen [Str+14] und [Str+15], aktualisiert für die Version 1.1 und ergänzt um aktuelle Erfahrungen.*

## 24.1 Einleitung

Es existiert eine Vielzahl von Unterstützungssystemen für die Programmierausbildung und für (automatische) Bewertungssysteme (vgl. Kapitel 2 bzw. [AM05; Iha+10; KJH16]). Es kann sogar angenommen werden, dass vermutlich jedes Institut für Informatik ein eigenes System zur Unterstützung des Übungsbetriebes entwickelt hat. Die Gründe für eine solche Entwicklung sind verschieden (vgl. Kapitel 2): Einerseits gibt es einen Trend zur Automatisierung, um auch größere Gruppen von Lernenden adäquat betreuen zu können, wobei viele Systeme dabei für einen konkreten Kontext entwickelt wurden. Andererseits sind forschungsgetriebene Entwicklungen zu nennen, aus denen innovative Unterstützungsansätze hervorgegangen sind. Daraus resultiert, dass es sich um abgeschlossene Systeme handelt, wobei viele Systeme zum Teil ähnliche Funktionen aufweisen (z. B. JUnit für Tests von Java-Programmen verwenden), sich aber in einigen Details (z. B. Softwarearchitektur, unterstützte Programmiersprachen sowie Art der Aufgaben,

Teile dieses Kapitels entstanden im Rahmen des Projekts eCULT, Teilvorhaben eAssessment, gefördert durch das Bundesministerium für Bildung und Forschung unter den Förderkennzeichen 01PL16066H und 01PL11066L. Die Verantwortung für den Inhalt dieses Kapitels liegt bei den Autoren und der Autorin.

der Bewertung oder des Feedbacks) unterscheiden. Aufgrund der speziellen, oftmals kontextbezogenen Anforderungen und aktueller Forschung in diesem Bereich ist auch nicht davon auszugehen, dass sich langfristig eines oder lediglich einige wenige Systeme global durchsetzen – unabhängig davon, ob einige Systeme eine größere Verbreitung erfahren.

Dennoch ist allen Systemen gemein, dass in irgendeiner Art und Weise Aufgaben erdacht und gestaltet werden müssen, die von den Lernenden bearbeitet werden sollen. Dabei muss neben didaktischen Aspekten vor allem darauf geachtet werden, dass keine Uneindeutigkeiten in der Aufgabenstellung entstehen. Dies gilt insbesondere für die Fälle, in denen Lösungen von Lernenden automatisch überprüft werden sollen (vgl. [AM05]). Hier muss mit größter Sorgfalt vorgegangen werden, da selbst kleinste Ungenauigkeiten in der Aufgabenstellung oder Fehler in Tests oder Musterlösungen zu großen Problemen führen können: Bei formativem Feedback können Lernende irritiert oder in eine falsche Richtung gelenkt und bei summativem Feedback im schlimmsten Fall falsche Noten vergeben werden. Dies sorgt für einen nicht zu unterschätzenden Aufwand für die Erstellung von Übungsaufgaben. In der Literatur finden sich Angaben zwischen mehreren Stunden und einer Personenwoche für die Erstellung einer einzigen Aufgabe (vgl. [KJH16]).

Trotz des teilweise enormen Aufwands für die Konzeption von Aufgabenstellungen, Musterlösungen und Tests gibt es bisher kaum etablierte Möglichkeiten, um Aufgaben und Testinhalte zwischen den Lehrenden auf einfache Art und Weise auszutauschen oder systemunabhängig zu nutzen. In diesem Kapitel wird das Austauschformat ProFormA vorgestellt, das diese Lücke eines fehlenden Austauschformats schließen soll und bereits von mehreren Systemen unterstützt wird. Für eine Übersicht über Anforderungen sowie weitere vorgeschlagene Austauschformate und deren Schwächen sei auf [Str+15] verwiesen.

## 24.2 ProFormA: Ein XML-Austauschformat für Programmieraufgaben

In diesem Abschnitt wird das XML-basierte Austauschformat ProFormA[1] in Version 1.1 beschrieben (in weiteren Publikationen finden sich Beschreibungen älterer Versionen: [Str+14] Version 0.9.1 und [Str+15] Version 0.9.4). Änderungen zwischen den Versionen werden im Folgenden kurz beschrieben. Das Austauschformat liegt sowohl als menschenlesbare Spezifikation mit Anmerkungen

1 Die genaue Spezifikation kann unter https://github.com/ProFormA/taskxml abgerufen werden.

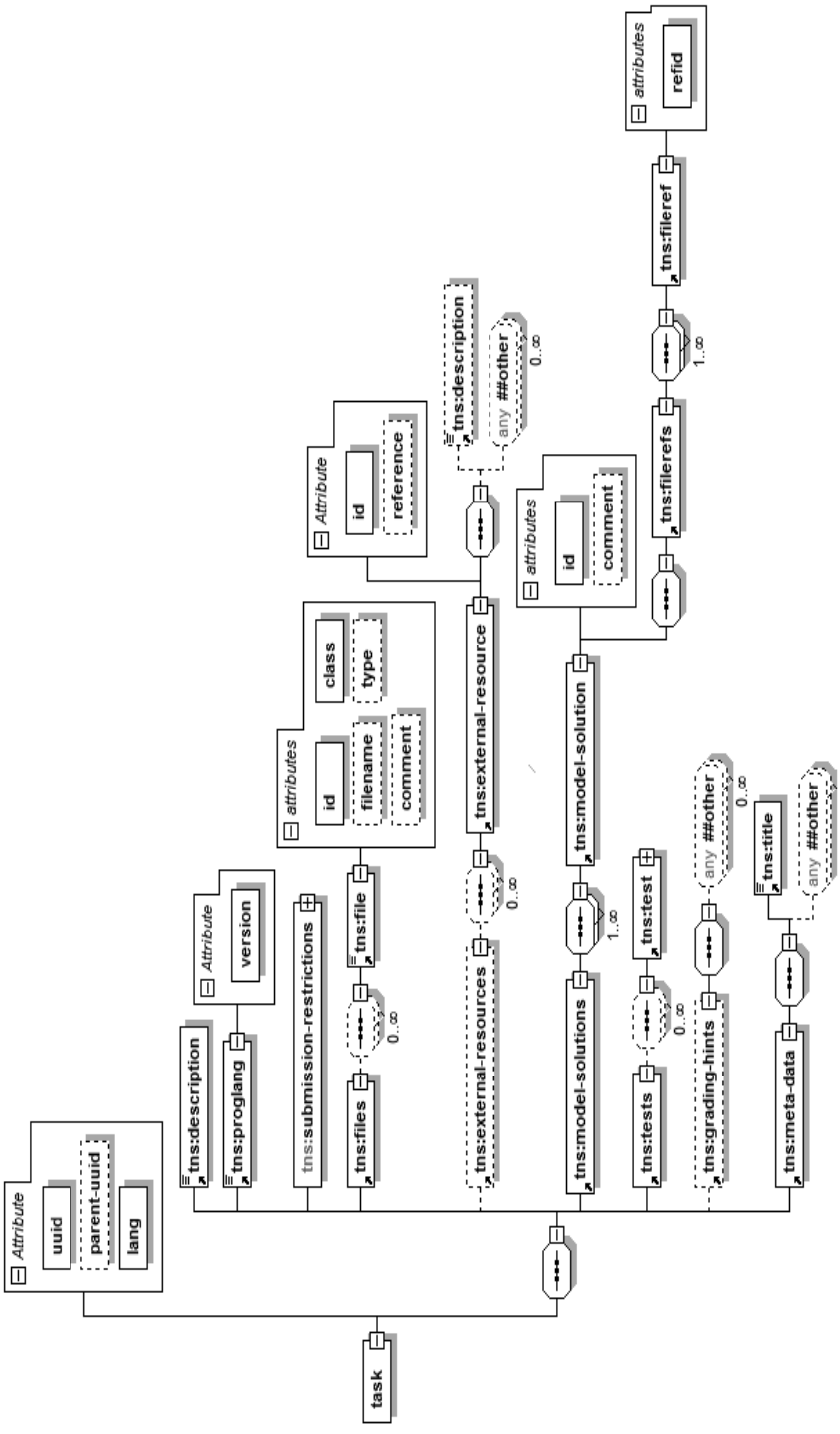


Abbildung 24.1: Strukturbaum des Austauschformats

zur genauen Bedeutung der einzelnen Tags als auch als maschinenlesbare XML-Schemadefinition vor, mit deren Hilfe erstellte Aufgaben auch automatisiert geprüft werden können.

Das Austauschformat spezifiziert für eine Aufgabe (*task*) einen Aufgabentext (Element */task/description*, inkl. Festlegung der Sprache zwecks Internationalisierung im Attribut *lang*), die Programmiersprache (Element */task/proglang* inkl. Version), zugehörige Dateien, technische Details zur Einreichung, Lösungsvorschläge, Metadaten und optional Hinweise zur Bewertung sowie Referenzen zu benötigten externen Ressourcen, die aufgrund ihrer Größe oder Struktur nicht in Verbindung mit der Aufgabe (z. B. als Anhang) ausgeliefert werden können. Mit Version 1.0.1 des ProFormA-Formats ist ein Universally Unique Identifier (UUID) zur eindeutigen Identifikation einer jeden im Austauschformat spezifizierten Aufgabe hinzugekommen. Dabei soll die UUID unter anderem eine Versionierung erleichtern, wobei bei modifizierten Aufgaben zusätzlich die UUID der originalen unmodifizierten Aufgabe als *parent-uuid* angegeben werden kann. Wie schon in vorherigen Versionen können zu einer Aufgabe mehrere Tests gehören, welche jeweils einen Testtyp (konkretes Bewertungsverfahren) und eine spezifische Testkonfiguration beinhalten. Tests werden in der Regel durch übliche Software-Engineering-Werkzeuge (Compiler, Unit-Tests, FindBugs, CheckStyle usw.) ausgeführt. Das Austauschformat selbst muss also im Wesentlichen die Bedingungen für die Ausführbarkeit der Tests festlegen, nicht aber den eigentlichen Testinhalt, welcher im Format der benutzten Werkzeuge gespeichert wird (z. B. skriptbasierte Blackbox-Tests oder XML-Konfigurationen für CheckStyle). Ein importierendes System kann folglich anhand der Testtypen entscheiden, welche Tests es direkt unterstützt und wie diese gegebenenfalls ausgeführt werden.

Abbildung 24.1 zeigt die Struktur des Austauschformats in der aktuellen Version 1.1. Dabei wird die Schachtelung der definierten Elemente und Attribute ersichtlich. Elemente, bei denen in der linken oberen Ecke drei stilisierte Linien zu sehen sind, können direkt mit Inhalten gefüllt werden (z. B. */task/description*). Optionale Elemente sind mit einer gestrichelten Linie umgeben dargestellt (z. B. */task/grading-hints*). XML-Sequenzen werden durch achteckige Kästen mit drei horizontal auf einer Linie befindlichen Punkten symbolisiert, wobei die Kardinalitäten direkt darunter angegeben sind, sofern diese von 1 abweichen. XML-Alternativen (xs:choice) werden ebenfalls durch achteckige Kästen mit drei vertikalen Punkten jedoch mit einem Auswahlzeiger (xs:choice, z. B. *submission-restrictions* in Abbildung 24.4) oder verbundenen horizontalen Linien (ebenfalls dort) dargestellt. Insbesondere sollen hier auch die Boxen mit „any ##other“ erwähnt werden: Das vorgeschlagene Austauschformat definiert lediglich das Grundgerüst, den gemeinsamen Nenner für Programmieraufgaben, und soll gleichzeitig

zum Beispiel für neue Programmiersprachen, Testverfahren und besondere Fähigkeiten von implementierenden Systemen erweiterbar sein. Zu diesem Zweck sind ähnlich zu Programmierframeworks sog. „Hotspots“ vorgesehen, an denen weitere Elemente aus anderen XML-Namespaces importiert und genutzt werden können – somit bleibt das Format für systemspezifische Konfigurationen und (neue) Bewertungsverfahren flexibel lokal erweiterbar. Zudem ist die vorgeschlagene Spezifikation versioniert (über die Namespace URI), um spätere Ergänzungen oder Änderungen der Spezifikation zu erlauben (die aktuelle Version trägt die URI „urn:proforma:task:v1.1“). Seit Version 0.9.4 des Formats müssen sämtliche Tags mit einem Namespace-Identifier voll qualifiziert werden. Diese Änderung wurde vorgenommen, da Parser für Aufgaben so einfacher implementiert werden können.

Listing 1 zeigt ein konkretes Minimalbeispiel einer Aufgabe im vorgeschlagenen Format, wie sie in einem typischen Einführungskurs in die Java-Programmierung zum Thema Rekursion oder Schleifen vorkommen kann: Die Berechnung der n-ten Fibonacci-Zahl.

```
<?xml version="1.0" encoding="UTF-8"?>
<p:task xmlns:p="urn:proforma:task:v1.1" lang="en" uuid="66217f7d-ca8e-4666-a6f8-934c79c24206">
    <p:description>Calculate the n-th Fibonacci number. As a reminder:
                   The first two Fibonacci numbers are 0 and 1.
                   The following numbers are the sum of the previous two numbers
                   (0, 1, 1, 2, 3, 5, 8, 13, 21...). The class should be named Fibonacci
                   and the method to be written fibonacci has an int as an input parameter.
    </p:description>
    <p:proglang version="1.4">java</p:proglang>
    <p:submission-restrictions>
        <p:files-restriction>
            <p:required filename="Fibonacci.java"/>
        </p:files-restriction>
    </p:submission-restrictions>
    <p:files>
        <p:file class="internal" filename="FibonacciTest.java" id="f1" type="embedded">
            import junit.framework.TestCase;
            public class FibonacciTest extends TestCase {
              public void testPos() { assertEquals(13, Fibonacci.fibonacci(7)); }
              public void testNull() { assertEquals(0, Fibonacci.fibonacci(0)); }
            }
        </p:file>
        <p:file class="internal" filename="Fibonacci.java" id="f2" type="embedded">
            public class Fibonacci {
              public int fibonacci(int i) {
              if (i &lt;= 0) return 0;
              else if (i == 1) return 1;
              return fibonacci(i - 2) + fibonacci(i - 1);}
            }
        </p:file>
    </p:files>
    <p:model-solutions>
        <p:model-solution id="m1">
            <p:filerefs><p:fileref refid="f2"/></p:filerefs>
        </p:model-solution>
    </p:model-solutions>
    <p:tests>
        <p:test id="t1">
            <p:title>Compilation test</p:title>
            <p:test-type>java-compilation</p:test-type>
            <p:test-configuration/>
        </p:test>
        <p:test id="t2">
            <p:title>Calculation test</p:title>
```

```
            <p:test-type>unittest</p:test-type>
            <p:test-configuration xmlns:u="urn:proforma:tests:unittest:v1.1">
                <p:filerefs>
                    <p:fileref refid="f1"/>
                </p:filerefs>
                <u:unittest framework="JUnit" version="3">
                    <u:entry-point>FibonacciTest</u:entry-point>
                </u:unittest>
            </p:test-configuration>
        </p:test>
    </p:tests>
    <p:meta-data>
        <p:title>Calculation of n-th Fibonacci number</p:title>
    </p:meta-data>
</p:task>
```

Listing 1: Beispielinstanz einer Programmieraufgabe

Zum besseren Verständnis wird im Folgenden genauer auf die zentralen Elemente des Austauschformats eingegangen.

### 24.2.1 Dateianhänge

Dateianhänge können entweder direkt in das XML-Dokument eingebunden oder über die Angabe des entsprechenden Dateinamens in einem ZIP-Archiv referenziert werden. Der letztere Fall ist vor allem für Binärdaten vorgesehen, damit das XML-Dokument nicht zu groß und unübersichtlich wird. Dabei ist lediglich vorgegeben, dass sich das XML-Dokument als „task.xml“ im Wurzelverzeichnis befinden muss, damit ein ZIP-Archiv korrekt als ProFormA-Aufgabe erkannt werden kann – die Organisation und Benennung weiterer Dateien ist nicht festgelegt.

Eine Datei, unabhängig ob es sich um eine Musterlösung, einen Test oder sonstiges handelt, wird durch ein *file*-Element repräsentiert, das unterhalb des Elements */task/files* eingefügt wird. Die Verwaltung der Dateien an einer zentralen Stelle soll zum einen für Übersichtlichkeit sorgen und zum anderen zur Vermeidung von Redundanzen beitragen. Für jede Datei muss eine innerhalb des XML-Dokuments eindeutige ID zur späteren Referenzierbarkeit (z. B. bei der Testdefinition) und eine Klasse (Attribut *class*) angegeben werden, die den Zweck und die Zugangsrechte beschreibt. Vorgesehen sind in diesem Zusammenhang Codevorlagen (template), Bibliotheken (library), Eingabedaten (inputdata), zusätzliche Informationen oder Anweisungen zur Bearbeitung einer Aufgabe (instruction), interne Dateien (z. B. Testtreiber oder Dateien zu einer Musterlösung, internal) sowie interne Bibliotheken (z. B. notwendige Bibliothek zur Ausführung eines Tests, internal-library, neu seit Version 0.9.4). „Intern“ bezieht sich dabei auf die Einschränkung, dass diese Dateien für Lernende grundsätzlich nicht zugänglich sind. Ob eine Datei direkt eingebunden ist oder im ZIP-Archiv referenziert wird, wird durch das Attribut *type* festgelegt. Zu jedem *file*-Element kann optional ein

Kommentar hinzugefügt werden (Attribut *comment*), um zusätzliche Informationen zur Datei anzugeben.

### 24.2.2 Musterlösungen

Für eine Aufgabe muss des Weiteren mindestens eine Musterlösung hinterlegt werden. Für diese doch recht strenge Anforderung gibt es mehrere Gründe: Zum einen sind diese technischer Natur, da es Systeme gibt, die eine Musterlösung benötigen, zum Beispiel um einen JUnit-Test zu übersetzen. Zum anderen hat dies auch praktische Gründe, da es eine Musterlösung zusammen mit der Aufgabenbeschreibung Lehrenden erlaubt die Schwierigkeit und Angemessenheit einer Aufgabe für den Einsatz im eigenen Kurs einzuschätzen.

Musterlösungen werden innerhalb des Elements */task/model-solutions* separat behandelt. Ein Element *model-solution* steht für eine konkrete Musterlösung, die aus einer Menge von Dateien bestehen kann (z. B. eine Musterlösung einer Java-Programmieraufgabe, bestehend aus mehreren .java-Dateien). Wie weiter oben bereits angedeutet, wird jede Datei, die zu einer Musterlösung gehört, in einem separaten *file*-Element definiert. Jedes dieser *file*-Elemente, wird über die Angabe seiner ID in einem */task/model-solution/filerefs/filref* -Element der jeweiligen Musterlösung zugeordnet.

Vor Version 0.9.4 konnte nur eine einzige Musterlösung angegeben werden. Zudem wurde noch nicht das bereits bestehende Konzept der *filerefs* genutzt, da Musterlösungen ursprünglich als separat angesehen wurden. Eine Änderung wurde vorgenommen, damit Redundanzen vermieden werden können, wenn zum Beispiel eine Datei zugleich Musterlösung und eine Eingabe für einen Test darstellt. Zur Angabe detaillierterer Informationen zu einer Musterlösung innerhalb eines *model-solution*-Elements lässt sich ebenfalls seit Version 0.9.4, wie bei einem *file*-Element, ein Attribut *comment* nutzen. Die Idee dahinter besteht darin, dass so unterschiedliche Musterlösungen menschenlesbar charakterisiert werden können (z. B. zur Unterscheidung von alternativen Lösungswegen, wie iterativ vs. rekursiv).

### 24.2.3 Tests und Bewertungsverfahren

Anzuwendende Tests oder Bewertungsverfahren werden innerhalb des Elements */task/tests* in einzelnen *test*-Elementen definiert (vgl. Abbildung 24.2). Neben der Angabe, welche Arten von Tests für eine Aufgabe zur Verfügung stehen (z. B.

Compile/Syntax- oder Unit-Tests, Element *test/test-type*) und der Angabe eines Titels (Element *test/title*), lässt sich die genaue Konfiguration eines Tests beschreiben (*test/test-configuration*). Zur Konfiguration eines Tests gehört, welche weiteren Dateien (z. B. Testtreiber) oder externen Ressourcen benötigt werden. Die für benötigte Dateien zu definierenden */files/file*-Elemente werden analog zu den Musterlösungen über *fileref* -Elemente referenziert. Darüber hinaus ist vorgesehen, dass direkt innerhalb eines *test-configuration*-Elements für jeden Testtyp ein eigener XML-Namespace genutzt werden kann, um testspezifische Parameter austauschen zu können (z. B. den Namen der aufzurufenden Testklasse für Unit-Tests, vgl. Listing 1).
Innerhalb des offiziellen ProFormA-Namensraumes sind aktuell die folgenden drei Testtypen vorgesehen:

**java-compilation** Der Java-Syntaxtesttyp benötigt keine weitere Testkonfiguration (vgl. Listing 1), da ein implementierendes System durch die abgegebenen Dateien und die referenzierten Dateien notwendige Bibliotheken automatisch laden kann.

**unittest** Dieser Testtyp ist seit Version 0.9.4 für allgemeine Unit-Tests vorgesehen. Vorher sollten für verschiedene Sprachen unterschiedliche Testtypen genutzt werden, was einen erhöhten Implementierungs- und vor allem auch Wartungsaufwand bedeutet hätte. Zur Konfiguration der Tests gibt es eine Spezifikation mit dem Namespace „urn:proforma:tests:unittest:v1.1“ (vgl. Listing 1). Dort ist ein einziges Element *unittest* spezifiziert (vgl. Abbildung 24.3), wo in den Attributen *framework* sowie *version* das zu verwendende Unit-Test-Framework (z. B. JUnit in Version 3) angegeben werden kann. Ferner bedürfen Unit-Tests der Angabe eines Einstiegspunkts oder Main-Klasse (Element *entry-point*). Zusammen mit den *filerefs* können die Unit-Tests ausgeführt werden.

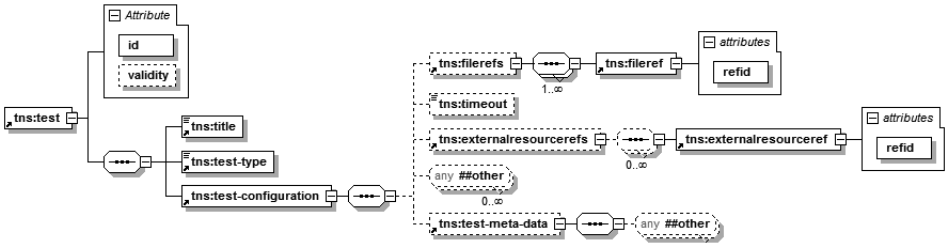


Abbildung 24.2: Strukturbaum des Elements */task/tests/test*

**regexptest** Dieser neueste spezifizierte Testtyp wird über das Element *regexp-test* des XML-Namespaces „urn:proforma:tests:regexptest:v0.9“ konfiguriert (vgl. Abbildung 24.3). Dieser Testtyp dient zur Überprüfung der Ausgaben einer Lernerlösung anhand von vorgegebenen regulären Ausdrücken. Dabei können mehrere reguläre Ausdrücke entweder als *regexp-allow* oder *regexp-disallow*-Elemente angegeben werden (inkl. weiterer Attribute um das Matching zu beeinflussen), die entweder alle zutreffen müssen beziehungsweise nicht zutreffen dürfen, um den Test zu bestehen. Wie die Lernerlösung ausgeführt werden soll, wird wie bei den oben genannten Tests über die Elemente *entry-point* und *parameter* für Kommandozeilenparameter festgelegt.

Zudem lassen sich innerhalb eines *test-configuration/test-meta-data*-Elements mit der Nutzung eines eigenen XML-Namespaces systemspezifische Metadaten (z. B. für Beschränkungen) für einen Test hinterlegen.
Grundsätzlich sollten Aufgabenspezifikationen in sich abgeschlossen sein, da nur auf diese Weise Aufgaben über einen langen Zeitraum ohne möglichen Verlust von Informationen oder Komponenten einsetzbar sind. Es haben sich jedoch Szenarien gezeigt, in denen die Nutzung von externen Ressourcen unvermeidbar ist, beispielsweise für Aufgaben, die automatisch bewertet werden sollen und dafür

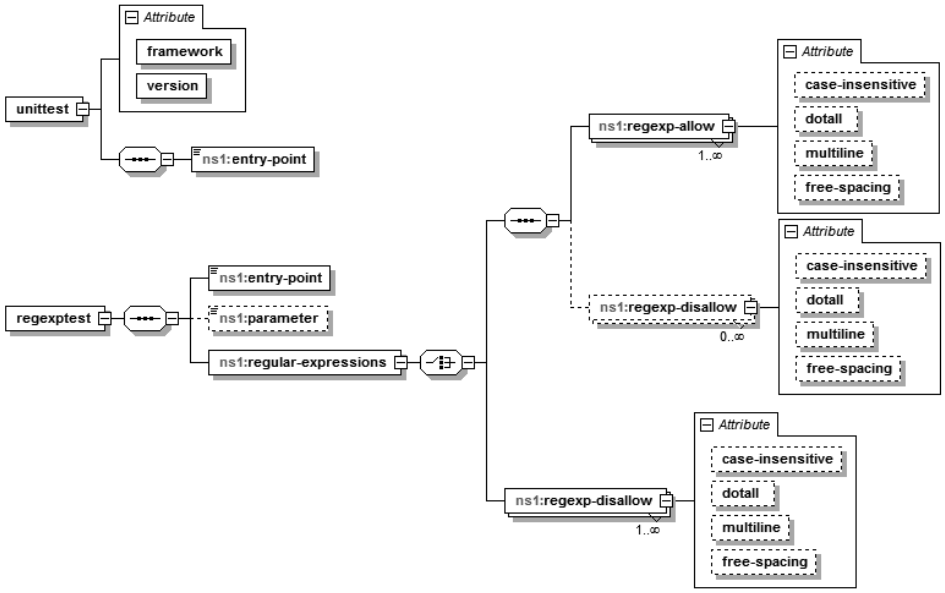


Abbildung 24.3: Strukturbäume der bisher spezifizierten Testtypen

sehr große Dateien als Abhängigkeiten benötigen, die nicht sinnvoll mit der Aufgabenspezifikation zusammengeführt werden können. Für diese Szenarien wurde das */task/external-resources* in Version 0.9.3 zur ProFormA-Spezifikation hinzugefügt, mit dem externe Ressourcen über eine eindeutige Referenz (*reference* Attribut) verknüpft werden können. Optional ist ein *description*-Element, in dem die Ressource näher beschrieben werden kann, und ein systemspezifischer XML-Namespace vorgesehen. Eine externe Ressource kann u. a. eine URL zu einem Datenbankabbild sein (z. B. ftp://ftp.fu-berlin.de/pub/misc/movies/database). Die Semantik und das genaue Format der Referenzen ist aktuell noch nicht genau spezifiziert, jedoch in dieser generischen Form integriert, um weitere Erfahrungen mit neuen Einsatzmöglichkeiten (vgl. Kapitel 23) sammeln zu können. Die Referenzierung externer Ressourcen, die innerhalb von *external-resource*-Elementen spezifiziert werden, erfolgt analog zu den *filerefs* unter Angabe ihrer jeweiligen ID in *externalresourcerefs/externalresourceref*-Elementen unterhalb von *test-configuration*-Elementen.

### 24.2.4 Abgabebeschränkungen

Das Element */task/submission-restrictions* bietet die Möglichkeit, Einschränkungen für die Einreichung von Lösungen zu definieren (vgl. Abbildung 24.4). In Version 0.9.1 hieß dieses Element noch *submission* und wurde zur Verbesserung des Verständnisses in nachfolgenden Versionen umbenannt. Zudem war es bis zur Version 0.9.4 lediglich möglich die maximale Größe (in Byte) der hochzuladenden Dateien festzulegen und über reguläre Ausdrücke Einschränkungen an den Dateinamen vorzunehmen. Mit Version 1.0.0 wurden diese Möglichkeiten deutlich feiner spezifiziert, wobei eine automatische Konvertierung sowohl zu älteren Versionen als auch zur Version 1.0.0 möglich ist. Gründe dafür bestanden darin, dass die Semantik der bisherigen Spezifikation sich als nicht klar genug herausgestellt hatte und Anforderungen von Systemen wie Praktomat (Limitierung der hochzuladenden Dateien über MIME-Type-Beschränkungen) und JACK (genaue Angabe von erforderlichen und optionalen Dateien, um im System zum Beispiel eine Abgabe auf Vollständigkeit prüfen zu können oder pro Datei ein eigenes Upload-Feld bereitzustellen) bisher nicht vollständig erfüllt wurden.

Beschränkungen oder Beschreibungen von abzugebenden Dateien können in einer von drei Formen erfolgen: *archive-restriction* (entspricht zu großen Teilen der Spezifikation aus Version 0.9.4), *files-restriction* oder *regexp-restriction* (entspricht fast vollständig der Spezifikation aus Version 0.9.4). Dabei kann in allen drei Fällen optional die maximale erwartete Größe (in Byte, *max-size*) einer

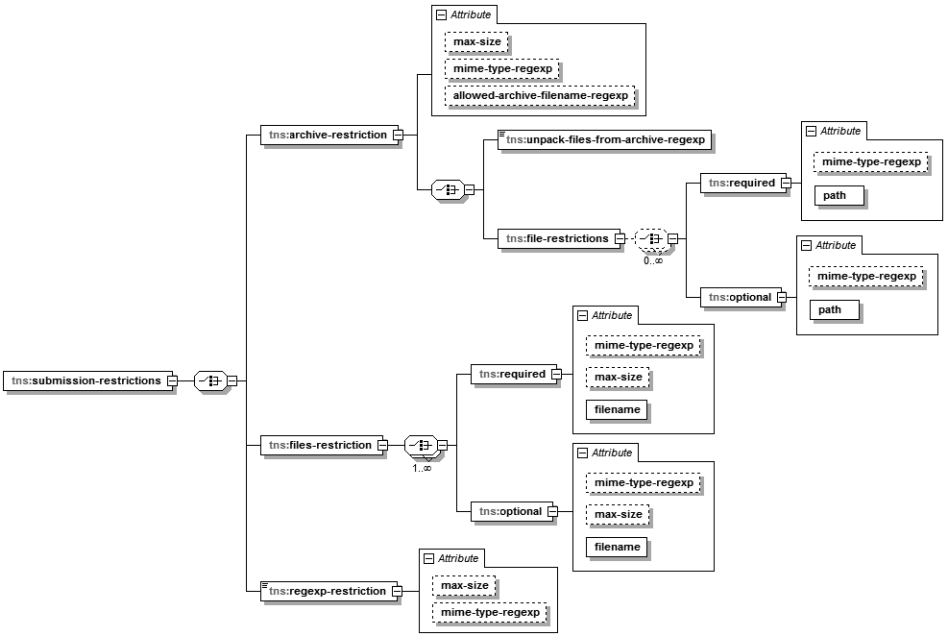


Abbildung 24.4: Strukturbaum des Elements */task/submission-restrictions*

zur Abgabe einer Lösung im System hochzuladenden Datei festgelegt werden. Natürlich kann ein implementierendes System eigene Größenbeschränkungen besitzen und den jeweils kleineren Wert anwenden. Die Möglichkeit, Einschränkungen für die Namen hochzuladender Dateien anzugeben, ist insbesondere für Java-Aufgaben relevant, da dort abhängig vom Klassennamen Anforderungen an den Dateinamen gestellt werden. Möchte ein Lernender eine Lösung einer Java-Aufgabe in Form eines Dateiuploads innerhalb eines Systems einreichen, so kann zunächst vom System verifiziert werden, ob der Name einer hochzuladenden Datei sämtliche Bedingungen eines vorgegebenen regulären Ausdrucks erfüllt beziehungsweise dem vorgegebenen Dateinamen entspricht. Ist dies nicht der Fall, so kann die Datei nicht hochgeladen und eine Warnung angezeigt werden. Somit lässt sich unter anderem sicherstellen, dass Tests nicht aufgrund eines falschen Dateinamens fehlschlagen.

Wird die Beschränkung *files-restriction* gewählt, so können die erwarteten Dateien einzeln spezifiziert werden. Dabei kann zwischen unbedingt erforderlichen (*required*) und optionalen Dateien (*optional*) einer Abgabe unterschieden werden. Es ist jeweils die Angabe des genauen Dateinamens (*filename*) sowie optional die maximale erwartete Dateigröße und eines regulären Ausdrucks als Anforderung an den MIME-Typ möglich. Dadurch kann insbesondere die Vollständigkeit einer

Abgabe frühzeitig erkannt werden. Für Systeme, welche die detaillierte Angabe von einzelnen Dateien nicht unterstützen und einen regulären Ausdruck erwarten (wie in früheren Versionen des Formats), kann dieser automatisch aus diesen präziseren Angaben generiert werden.

Die Beschränkung *regexp-restriction* erlaubt es über einen regulären Ausdruck Bedingungen an die Namen hochzuladender Dateien zu stellen, wobei zusätzlich mit Version 1.0 über ein Attribut auch Bedingungen an den MIME-Typ gestellt werden können.

Für komplexere Aufgaben, die zum Beispiel als Archiv abgegeben werden sollen, können über das Element *archive-restriction* Bedingungen an hochzuladende Archive festgelegt werden. Neben der Vorgabe eines erlaubten Archivnamens (Attribut *allowed-archive-filename-regexp*) und einer maximalen Archivgröße (Attribut *max-size*) können zusätzlich die Namen der automatisch zu extrahierenden Dateien entweder über einen regulären Ausdruck (Element *archive-restriction/unpack-files-from-archive-regexp*) oder der genauen Angabe von Dateipfaden innerhalb des hochzuladenden Archivs (Element *archive-restriction/file-restrictions*) angegeben werden (vgl. Element */task/submission-restrictions/files-restriction*).

### 24.2.5 Bewertungsschema und Metadaten

Für eine Aufgabe kann ein Bewertungsschema (Element */task/grading-hints*) sowie weitere Metadaten (Element */task/meta-data*) angeben werden. Innerhalb des Elements *meta-data* ist lediglich ein Element für die Angabe eines Titels für die Aufgabe verpflichtend vorgesehen (*title*). Weiterhin kann ein eigener XML-Namespace für systemspezifische Metadaten genutzt werden. Metadaten zur Kategorisierung wurden bisher bewusst nicht in das Format aufgenommen, da es für Metadaten zum einen etablierte Standards gibt (vgl. IEEE LOM, Dublin Core) und zum anderen diese oft nicht oder nur halbherzig benutzt werden [God04].

Analog zum Element *test-meta-data* kann im Element */task/grading-hints* ein eigener XML-Namespace für systemindividuelle Angaben zu Bewertungsschemata benutzt werden. Die Spezifikation von Bewertungsschemata wurde im ProFormA-Austauschformat bisher ebenfalls explizit ausgenommen, da diese oftmals sehr von den Vorstellungen und Vorlieben der Institutionen beziehungsweise Lehrenden abhängen (eine Untersuchung von [CR13] bestätigt eine sehr hohe Heterogenität). Dennoch können die Angaben von weiteren Lehrenden gelesen und eventuell adaptiert werden – jedoch in der Regel nicht automatisch.

Insgesamt können durch den expliziten Einsatz von individuellen XML-Namespaces, die nicht zur offiziellen ProFormA-Spezifikation gehören, aufgaben- oder

testbezogene Attribute spezifiziert werden, die für ein bestimmtes System benötigt werden, für andere Systeme aber nicht relevant sind. Systemindividuelle Angaben gehen dadurch beim Export nicht verloren. Ein vollständiger Export einer Aufgabe in das Austauschformat mit anschließendem verlustfreien Reimport in dasselbe System ist somit prinzipiell möglich. Wird eine aus einem System exportierte Aufgabe in ein anderes System importiert, so ist sichergestellt, dass nur relevante Daten importiert werden, die auch vom letztgenannten System berücksichtigt und verarbeitet werden können.

## 24.3 Erfahrungen mit dem Austauschformat und Diskussion

In diesem Abschnitt werden die aktuelle Unterstützung in verschiedenen Systemen und bisherige Erfahrungen mit dem Austauschformat beschrieben, Hinweise für Implementierungen gegeben sowie mögliche Probleme und Lösungen diskutiert.

### 24.3.1 Aktuelle Unterstützung von ProFormA

Die ProFormA-Spezifikation wird bereits seit ungefähr vier Jahren von mehreren Systemen (GATE (vgl. Kapitel 13), JACK (vgl. Kapitel 9), Graja (vgl. Kapitel 11) und einer Praktomat-Version (vgl. Kapitel 10)) zumindest zum Export von Java-Aufgaben unterstützt. Das System aSQLg (vgl. Kapitel 12) unterstützt die ProFormA-Spezifikation seit 2017 für SQL-Aufgaben.

Neben dem Export wurde für drei der aufgeführten Systeme (GATE, JACK, Praktomat) zudem ein Import implementiert. In diesem Zusammenhang wurden für diese Systeme systemübergreifende Importe (Export aus einem System und Import in ein anderes System) und systeminterne Importe (Export aus einem System und Reimport in dasselbe System) zu Testzwecken durchgeführt. Ein Export ist prinzipiell aus allen aufgeführten Systemen möglich. Beim JACK-System werden jedoch die notwendigen Musterlösungen zum aktuellen Zeitpunkt noch nicht exportiert (vgl. Abschnitt 24.2.2). GATE und Praktomat erzeugen vollständig valide XML-Dokumente gemäß Version 1.0.1 der Spezifikation. Ein Import ist sowohl von Version 0.9.4 als auch 1.0.1 in beiden Systemen möglich. Unterstützung für die Version 1.1 ist schon oder wird aktuell implementiert.

Ein systeminterner und systemübergreifender Import ist grundsätzlich für jedes der drei Systeme möglich. Die vollständige Ausführbarkeit einer importierten

Aufgabe, die zuvor aus einem anderen System exportiert wurde, ist jedoch nicht immer gewährleistet. Dies ist zum einen durch die unterstützten Bewertungsverfahren und zum anderen durch besondere Systemfeatures begründet. Beispielsweise werden alle drei Systeme für Java-Programmieraufgaben eingesetzt. GATE und Praktomat unterstützen beide JUnit-Tests – zwischen diesen beiden Systemen lassen sich daher Aufgaben samt Syntax- und JUnit-Tests vollständig austauschen und ausführen. Das JACK-System hingegen unterstützt keine JUnit-Tests, sondern setzt auf andere Bewertungsverfahren, die zurzeit exklusiv im JACK-System Anwendung finden. Im GATE-System findet sich (noch) keine vollständige Unterstützung für RegExp-Tests; sie erlauben bisher nur das Matching von korrekten Lösungen (*not-allowed-regexp* fehlt). Dennoch lassen sich durch die Möglichkeit systemspezifische Elemente in Form von Metadaten zu exportieren (Elemente */task/meta-data* und */task/tests/test/test-configuration/test-meta-data*), wie weiter oben bereits erwähnt, wesentliche Elemente von Programmieraufgaben systemübergreifend austauschen und zugleich systeminterne Importe verlustfrei durchführen.

### 24.3.2 Systemübergreifender Austausch in der Praxis

Wie im vorherigen Abschnitt beschrieben, ist ein Austausch von Aufgaben zwischen den Systemen GATE, JACK und Praktomat seit mehreren Jahren mithilfe der ProFormA-Spezifikation möglich. Einsetzende Institutionen besitzen zudem recht gut gefüllte Pools von Programmieraufgaben mit teilweise sehr ausführlichen Tests (GATE an der TU Clausthal: ca. 250 Java-Aufgaben, JACK an der Universität Duisburg-Essen: ca. 70 Java-Aufgaben, Praktomat an der Hochschule Ostfalia: 60 Aufgaben in Java, Python und einer mathematischen Programmiersprache). Dennoch wurde ein hochschulübergreifender Austausch bisher nur relativ selten praktiziert – auch wenn allgemeines Interesse daran besteht, neue, bewährte Aufgaben zu nutzen. Hauptgründe sind sicherlich, dass Lehrende bisher keinen Zugriff auf die Aufgabenpools der anderen Institutionen besitzen und sich folglich keinen Überblick über potenziell passende Aufgaben verschaffen können. Dies erfordert bisweilen einen persönlichen Kontakt, wobei Aufgaben schließlich über E-Mail ausgetauscht werden.

Zur Verbesserung des Austausches wäre daher eine Möglichkeit wünschenswert, direkt aus dem lokal genutzten System neue Aufgaben suchen und importieren zu können. Dies würde zudem die Attraktivität von ProFormA für andere Systeme erhöhen und sich wahrscheinlich auch wieder positiv auf den Aufgaben-

bestand sowie die Qualität der Lehre durch Nutzung bewährter Aufgaben auswirken.

Lösungsmöglichkeiten bestehen im Aufbau eines dezentralen Verbundes wie es zum Beispiel beim LON-CAPA-Verbund der Fall ist oder eines zentralen Repositories. Beide Ansätze haben charakteristische Vor- und Nachteile. In einem dezentralen Verbund müssten in allen Systemen Peer-to-Peer-Technologien implementiert und genutzt werden – ein Aufwand, der für jedes sonst unabhängige System durchgeführt und auch aktuell gehalten werden muss. Zudem ist es erforderlich, dass alle Server gut an das Internet angebunden und stets erreichbar sind. Dafür müssten jedoch keine Aufgaben auf einem „fremden“ System gespeichert werden und lokale Statistiken wären möglich. Ein zentrales Repository würde lediglich die Implementierung einer Schnittstelle in allen teilnehmenden Systemen erfordern. Die Schnittstelle könnte wiederum versioniert sein, so dass Änderungen nur sehr selten notwendig werden. Statistiken könnten global erhoben werden, jedoch müsste eine langfristige Bereitstellung sowie Finanzierung des Repositories sichergestellt werden. In beiden Fällen ist es zudem von enormer Bedeutung, dass ein Zugriff auf die vollständigen Aufgabenspezifikationen für Lernende ausgeschlossen ist (vgl. LON-CAPA [Kor+08]), da diese jeweils eine Musterlösung und auch die Tests beinhalten. Dadurch muss darüber hinaus auch sichergestellt werden, dass langfristig ebenfalls eine Instanz existiert, die über Aufnahme oder Zugriff entscheiden kann.

Nach Abwägung der Vor- und Nachteile, wird der Aufbau eines Repositories für Aufgaben angestrebt. Hierfür muss nicht unbedingt ein eigenes Repository aufgesetzt werden, sondern grundsätzlich kommt auch eine Mitnutzung bestehender Repositories in Betracht, wobei dabei sicherlich auch auf weitere verbreitete Standards wie zum Beispiel IMS Content Packaging[2] zurückgegriffen werden kann. Leider konnte bisher keine zufriedenstellende Lösung in Bezug auf die Mitnutzung unter Erreichung der oben genannten Anforderungen gefunden werden. Derweil laufen Gespräche mit von Universitäten unabhängigen Vereinen, die sich der Förderung von E-Learning widmen, um sowohl die Auswahlinstanz als auch das Repository dort anzusiedeln.

### 24.3.3 Authoring-Funktionalität

Eigentlich wird kein spezieller Editor für das Format benötigt, da jedes existierende System bereits eigene Authoring-Funktionalitäten bereitstellt. Dennoch kann ein systemunabhängiger Editor zentrale Vorteile bieten:

2 http://www.imsglobal.org/content/packaging/

- stete Unterstützung der aktuellsten ProFormA-Version durch unabhängige Entwicklung,
- Validierung, Einlesen und Bearbeiten von Aufgaben,
- Konvertierung von Aufgaben aus und in verschiedene Formatversionen.

Daher wurde ein eigener quelloffener Editor entwickelt[3] (vgl. Abbildung 24.5). Dieser Editor ist komplett in JavaScript geschrieben. Dadurch kann dieser sowohl systemunabhängig genutzt als auch relativ leicht in bestehende Systeme integriert werden. Aktuell wird der Editor produktiv an der Hochschule Ostfalia zur Erstellung von Aufgaben eingesetzt, weil dort der Praktomat, der die eigentliche Auswertungsfunktion für die Programmieraufgaben bereitstellt, komplett in das dort eingesetzte LMS LON-CAPA integriert ist und für die Lehrenden unsichtbar sein soll (vgl. Kapitel 22). Anstatt LON-CAPA um Authoring-Funktionalität für Programmieraufgaben zu ergänzen, erschien es sinnvoller, den Editor gleich so zu konzipieren, dass er in verschiedene Systeme integriert werden kann. Der Editor sieht vor, dass für jedes System eine JavaScript-Datei angelegt wird, welche die Metadaten und speziellen Namespaces, die nur von dem System benötigt werden, bereitstellt. Für die Benutzung mit LON-CAPA produziert der Editor außer dem Austauschformat auch noch eine zusätzlich von LON-CAPA benötigte Konfigurationsdatei. Eine derart vollständige Unterstützung fehlt im Editor noch für die anderen Systeme.

### 24.3.4 Erweiterbarkeit der Spezifikation

Beim Entwurf der Spezifikation wurden die Anforderungen unterschiedlicher Systeme (u. a. GATE (vgl. Kapitel 13), JACK (vgl. Kapitel 9), einer Praktomat-Version (vgl. Kapitel 10) und ViPS (vgl. Kapitel 14) sowie grundsätzlich 29 weitere) an Aufgabenspezifikationen beachtet (vgl. [Str+15]). Dadurch ist sichergestellt, dass das Format prinzipiell nicht nur die Anforderungen eines einzelnen Systems und lediglich die dort unterstützten Programmiersprachen implementiert, wie es für die meisten bestehenden Formate der Fall ist, sondern eine Obermenge der Anforderungen umfasst (vgl. [Str+15]). Zudem wird die vorgeschlagene Spezifikation nicht als festes „one-size-fits-all“-Format angesehen und systemspezifische Aspekte werden nicht ignoriert, sondern Erweiterungen einzelner Systeme oder neuer Testverfahren können über eigene, frei definierbare XML-Namespaces eingebettet werden. Dies soll zudem die Hürde senken ProFormA zu unterstützen, da auf diese Weise systemspezifische Aspekte nicht ausgelassen werden müssen.

3 https://github.com/ProFormA/formatEditor

Eine Erweiterbarkeit (über XML-Namespaces) birgt jedoch grundsätzlich auch die Gefahr einer „Zerfaserung“ des Austauschformats, wenn elementare Aspekte dort mehrfach oder unterschiedlich definiert werden. Diesem Aspekt wurde zum einen dadurch Rechnung getragen, dass wesentliche Aspekte bereits im vorgeschlagenen Format selbst definiert wurden. In der Praxis hat sich diese Annahme bisher bestätigt. Zum anderen können durch die Versionierung der Spezifikation später noch Änderungen und Erweiterungen vorgenommen werden, um „verbreitete“ Aspekte mit in das Format aufzunehmen. Doppeldefinitionen traten bisher nicht auf. Ein Grund dafür ist sicherlich, dass bisher alle Maintainer von Systemen mit ProFormA-Unterstützung an der Entwicklung von ProFormA (zumindest am Rande) beteiligt sind und mögliche Fälle direkt zur Diskussion stellen. Unabhängig davon haben sich seit der in [Str+14] veröffentlichten Version 0.9.1 einige Verbesserungsmöglichkeiten gezeigt (vgl. vorheriger Abschnitt). Zum Beispiel wurden Aspekte erweitert, die sich in älteren Versionen der Spezifikation nicht abbilden ließen (z. B. Modellierung von einzelnen Dateien bei der Abgabe, auch zur Vollständigkeitsprüfung) oder der Verbesserung des Verständnisses dienen (z. B. Umbenennungen) beziehungsweise die Handhabung vereinfachen (bspw.

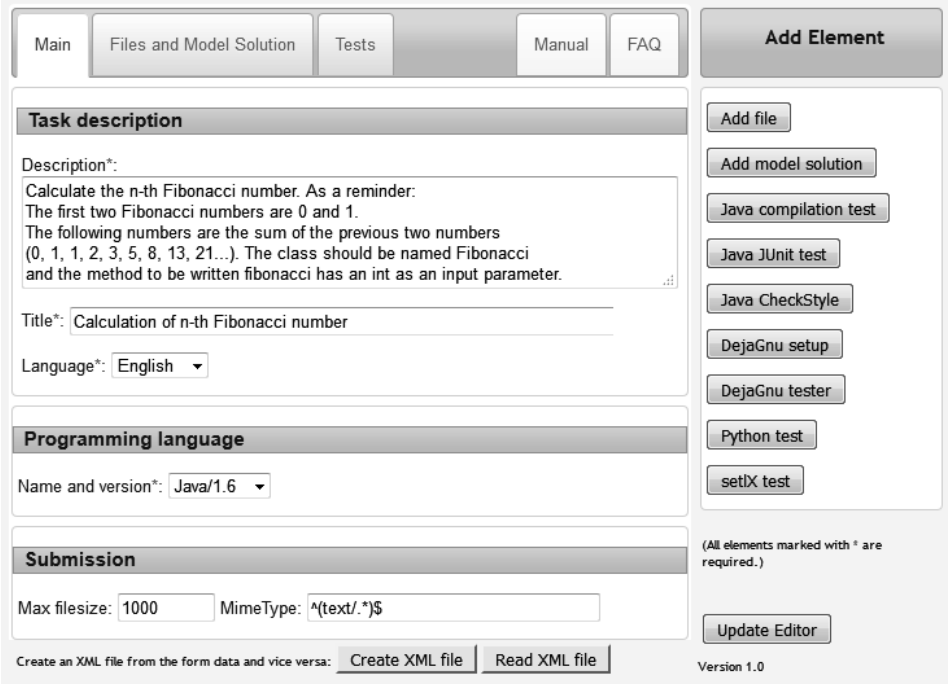


Abbildung 24.5: Benutzeroberfläche des ProFormA-Editors

Namespace-Präfixe für XML-Elemente). Bei frühen Versionen ergaben sich häufiger und größere Änderungen – da zu diesem Zeitpunkt auch nur recht wenige Aufgabenspezifikationen vorlagen, waren auch größere Veränderungen ohne Bedenken möglich – bei Versionen ab 0.9 hat sich bereits eine gewisse Stabilität mit jeweils ungefähr einem Jahr zwischen neuen Versionen gezeigt. Die Verwaltung von Änderungen an der Spezifikation mit Hilfe von Git (auf GitHub) hat sich allgemein als sehr hilfreich erwiesen – nicht nur um Änderungen verfolgen zu können, sondern auch um Vorschläge zu diskutieren und zu entwickeln.

Versionierung verhindert jedoch nicht, dass sich bei Änderungen an der Spezifikation notwendige Anpassungen an implementierenden Systemen ergeben, um Aufgaben importieren zu können, die mit der neuen Version beschrieben wurden. Es ist aber zu beachten, dass dies nur für Aufgaben gemäß der neuen Spezifikation gilt und ältere Aufgaben, bei entsprechender Unterstützung, weiterhin genutzt werden können. Es wird empfohlen bei Erweiterungen die Unterstützung für ältere Versionen nicht zu deaktivieren. Zudem sind für hinzukommende Testtypen oder andere Erweiterungen, die von einem System nicht unterstützt werden, keine Änderungen notwendig, so dass lediglich bei bestimmten Veränderungen an der ProFormA-Kernspezifikation und unterstützten Testtypen Änderungen notwendig werden.

Da es bisher kein Repository gibt, die Aufgaben größtenteils in den Systemen verwaltet und bei Bedarf von dort exportiert werden, ist es zurzeit ausreichend bei einer neuen Implementierung die aktuellste Version der Spezifikation zu unterstützen. Konvertierungen sind aktuell mit dem ProFormA-Editor möglich. Aufgrund der Nutzung von XML ist es aber auch möglich, Aufgaben mit Hilfe einer XSL-Transformation automatisch zu transformieren – bei der Nutzung eines zentralen Repositories könnte dies transparent vor dem Import in ein System durchgeführt werden.

Parser für das ProFormA-Format liegen für Praktomat in Python und GATE in Java vor. Da beide Systeme unter einer Open-Source-Lizenz veröffentlicht wurden, kann auf diese Implementierungen als Ausgangspunkt zurückgegriffen werden. Um eine eigene Implementierung der ProFormA-Spezifikation testen zu können, werden unter https://github.com/ProFormA/examples Beispielexporte aus verschiedenen Systemen bereitgestellt – zukünftig auch in verschiedenen Versionen der Spezifikation.

### 24.3.5 Weitere Nutzungsmöglichkeiten

Das Format erlaubt grundsätzlich neben dem Austausch von Aufgaben zwischen reinen Assessment-Systemen auch die Nutzung als Beschreibung für Aufgaben zwischen einem LMS und einem Evaluationssystem (serviceorientierte Architektur oder einer Middleware): Ein LMS kann die Verwaltung sowie Darstellung der Aufgaben übernehmen und zur Evaluation die studentischen Einreichungen zusammen mit der Aufgabenbeschreibung an einen Evaluationsservice beziehungsweise eine durch Plugins erweiterbare Middleware, die mehrere Programmbewertungssysteme einbinden kann, schicken (vgl. [Str+15] sowie Kapitel 22 und 23). An der Hochschule Ostfalia wird die Verbindung vom Praktomat mit LON-CAPA schon auf diese Art realisiert (vgl. Kapitel 22). Um einen synchronen Austausch auch zwischen anderen Systemen zu ermöglichen, sind aber noch eine kollaborativ erstellte Spezifikation und die Implementierung von Schnittstellen auch in anderen Systemen erforderlich. Dies ist derzeit bereits in Planung. Insbesondere erfordert dies auch die Spezifikation eines Antwortformats.

## 24.4 Zusammenfassung und Ausblick

In diesem Artikel wurde eine Spezifikation *ProFormA* zum systemübergreifenden Austausch von Programmieraufgaben vorgestellt. Dabei handelt es sich nicht um eine „one-size-fits-all"-Spezifikation für vorher festgelegte Sprachen und Testtypen, sondern um ein durch XML-Namespaces erweiterbares Format. Damit bietet das Format eine gute Grundlage zur Erhöhung der Interoperabilität zwischen verschiedenen Systemen – grundsätzlich könnte das Austauschformat von allen 33 im Review von [Str+15] enthaltenen Systemen genutzt werden. Ein Export und Import von Aufgaben mit der vorgeschlagenen Spezifikation ist bereits aus mehreren Systemen möglich. Ebenso lassen sich wesentliche Elemente von Aufgaben, die in das Austauschformat exportiert wurden, in diese Systeme importieren. Ein systemübergreifender Austausch von Aufgaben unter Verwendung des vorgeschlagenen Austauschformats ist durchführbar.

Die vorgeschlagene Spezifikation wird bereits seit ungefähr vier Jahren von mehreren Systemen unterstützt. Dabei wurden viele Erfahrungen gewonnen, die auch wieder in die Spezifikation eingeflossen sind. Zudem ist in weiteren Systemen eine Unterstützung für das ProFormA-Format geplant. Ebenfalls haben sich neue Nutzungsszenarien für ProFormA ergeben: Statt lediglich Aufgaben zwischen Lehrenden auszutauschen, kann das Format auch dazu genutzt werden die Kommunikation zwischen LMS und Gradern zu vereinfachen [Str+15].

Anhand der Versionsgeschichte (vgl. Abschnitt 24.2) lässt sich gut erkennen, dass die Verfeinerung, Verbesserung und Erweiterung der Spezifikation keineswegs abgeschlossen ist. Insbesondere gibt es einige offene Punkte (z. B. parametrisierte Aufgaben, Zusammenfassen von Aufgaben und weitere Aspekte hinsichtlich Sicherheit und Laufzeitbeschränkungen von Tests) und die Spezifikation weiterer oft genutzter Testtypen, die noch in das offizielle ProFormA-Format einfließen sollen. Unabhängig von der Arbeit der Spezifikation wird für einen einfachen Austausch der Aufbau eines Repositories für Aufgaben angestrebt, so dass Aufgaben zum Beispiel direkt aus einem System heraus gesucht und importiert werden können, um so den Austausch weiter zu vereinfachen und zu fördern.

## Literatur für dieses Kapitel

[AM05] Kirsti M. Ala-Mutka. „A Survey of Automated Assessment Approaches for Programming Assignments“. In: *Computer Science Education* 15.2 (2005), S. 83–102. DOI: 10.1080/08993400500150747.

[CR13] Julio C. Caiza und José María del Álamo Ramiro. „Programming assignments automatic grading: review of tools and implementations“. In: *7th International Technology, Education and Development Conference (INTED2013)*. 2013, S. 5691–5700.

[God04] Jean Godby. „What do application profiles reveal about the learning object metadata standard?“ In: *Ariadne* 41 (2004). URL: http://www.ariadne.ac.uk/issue41/godby/.

[Iha+10] Petri Ihantola u. a. „Review of Recent Systems for Automatic Assessment of Programming Assignments“. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli Calling '10. ACM, 2010, S. 86–93. DOI: 10.1145/1930464.1930480.

[KJH16] Hieke Keuning, Johan Jeuring und Bastiaan Heeren. *Towards a Systematic Review of Automated Feedback Generation for Programming Exercises – Extended Version*. Techn. Ber. UU-CS-2016-001. Department of Information und Computing Sciences, Utrecht University, März 2016. URL: http://www.cs.uu.nl/research/techreps/repo/CS-2016/2016-001.pdf.

[Kor+08] Gerd Kortemeyer u. a. „Experiences using the open-source learning content management and assessment system LON-CAPA in intro-

ductory physics courses“. en. In: *American Journal of Physics* 76.4 (2008).

[Str+14] Sven Strickroth u. a. „Wiederverwendbarkeit von Programmieraufgaben durch Interoperabilität von Programmierlernsystemen“. In: *DeLFI 2014 – Die 12. e-Learning Fachtagung Informatik*. Bd. 233. LNI. GI, 2014, S. 97–108.

[Str+15] Sven Strickroth u. a. „ProFormA: An XML-based exchange format for programming tasks“. In: *eleed* 11.1 (2015). URL: https://eleed.campussource.de/archive/11/4138.