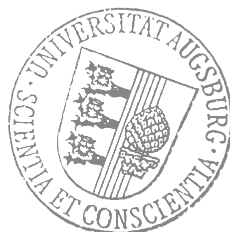
Universität Augsburg
Fakultät für Angewandte Informatik

Automatisiertes strukturelles und funktionales Testen von Fremdcode in einer sicheren Umgebung

UniBene: Ein System zum automatisierten Testen

Adrian Klein

Bachelorarbeit in Informatik und Informationswirtschaft



Erstgutachter: Prof. Dr. Wolfgang Reif

Zweitgutachter: Prof. Dr. Martin Wirsing

Abgabedatum: 9. September 2008

Aufgabensteller: Prof. Dr. Martin Wirsing

Betreuer: Andreas Schroeder

Erklärung

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Augsburg, den 9. September 2008 Adrian Klein

Zusammenfassung

Insbesondere in der Informatik sind elektronische Abgabesysteme mit Workflow-Unterstützung sehr beliebt, in denen Studenten im Wochenrhythmus ihre Lösungen zu den gestellten Übungsaufgaben abgeben können.

Häufig sollen Programme als Teil der Lösungen abgegeben werden. Auch wenn die gewünschte Struktur und Funktionalität dieser Programme in der Aufgabenstellung genau beschrieben wird, unterbleibt meist die automatische Überprüfung dieser Eigenschaften durch das Abgabesystem. Dies hat unvollständige und uneinheitliche Abgaben zur Folge, die nur mit hohem manuellen Aufwand zu korrigieren sind.

In dieser Arbeit wird eine Ergänzung für das Programmsystem UniWorX beschrieben, die den gesamten Prozess von der Aufgabenstellung durch den Übungsleiter, bis hin zur Abgabe durch den Studenten und der anschließenden Korrektur unterstützt. Insbesondere wird die Einhaltung struktureller Vorgaben bereits unmittelbar bei der Abgabe kontrolliert, und als Vorbereitung für die Korrektur werden vorbereitete Unit-Tests automatisch ausgeführt, sowie die Ergebnisse derselben den Korrektoren zusammen mit der Abgabe zur Verfügung gestellt.

Auf diese Weise profitieren Aufgabensteller, Studenten und Korrektoren gleichermaßen von diesem verbesserten Prozess.

Abstract

Especially in computer science departments at universities electronic submission system are very popular, which accept the students' solutions of the weekly exercises.

Often programs are part of these solutions. While the desired structure and functionality of these programs is usually precisely defined in the assignments, it is not uncommon that the submission system itself does not verify these requirements automatically. This leads to incomplete and inconsistent submissions, which can only be verified with considerable manual effort.

This work describes an extension of the submission system UniWorX, which will support the whole submission process, from the composition of the exercises to the submission and correction of the students' solutions.

Immediately after submission the structure of the solution will be matched against a given specification, and provided unit tests for this exercise will be executed automatically. Eventually the results will be delivered to the correctors as an addition to the original solutions.

Composers of assignments, students and correctors will profit from this improved submission process all alike.

Inhaltsverzeichnis

1 UniBene und UniWorX **1**

1.1 UniWorX 1

1.2 Automatisiertes Testen 2

1.3 Umsetzung 2

2 Methodik **5**

2.1 Entwicklungsprozess 5

2.2 Projektplanung 6

3 Anforderungen **9**

3.1 Ziele 9

3.2 Abgabeprozess 10

3.3 Funktionalität 11

4 Design und Evaluation **15**

4.1 Überblick 15

4.2 Spezifikationseditor 16

4.3 UniWorX-Integration 19

4.4 Validierungsserver 22

5 Ergebnisse und Ausblick **27**

5.1 Methodik 27

5.2 Anforderungen 27

5.3 Erweiterungsmöglichkeiten 28

5.4 Fazit 29

Literatur **31**

1 UniBene und UniWorX

UniWorX ist ein typisches Abgabesystem, das am Institut für Informatik der LMU bevorzugt eingesetzt wird: Es bietet zwar Übungsleitern, Studenten und Korrektoren eine ausgefeilte Workflow-Unterstützung für die wöchentlichen Abgaben der Studenten. Allerdings bietet es darüber hinaus keinerlei Möglichkeiten, genauer zu spezifizieren oder zu validieren, was im Einzelnen abgegeben werden soll. Das in dieser Arbeit beschriebene System UniBene, stellt daher eine Ergänzung zu UniWorX dar, die genau dies leisten soll. Durch eine nahtlose Integration von UniBene mit UniWorX ist gewährleistet, dass der bisherige Abgabeprozess beibehalten werden kann. Bevor also UniBene selbst betrachtet werden kann, ist es unabdingbar sich UniWorX genauer anzuschauen, da es den äußeren Kontext für UniBene bildet.

1.1 UniWorX

UniWorX ist eine in Java geschriebene Webanwendung, die in dem Servlet-Container Tomcat [BD03] läuft. Als darunter liegendes Framework wird Struts [Cav04] verwendet.

Der Workflow des Übungsbetriebs wird von UniWorX folgendermaßen unterstützt: Übungsleiter können Übungen anlegen, für welche sich dann Studenten anmelden können. Zu den regelmäßigen Übungsblättern können die Studenten ihre Lösungen bis zur Abgabefrist als Zip-Archiv hochladen. Anschließend werden die eingereichten Lösungen an die Korrektoren verteilt, und die fertige Korrektur wird schließlich den Studenten als Feedback zur Verfügung gestellt.

Sind Programme als Lösung gefordert, wird in den Aufgabenstellungen in der Regel genau beschrieben, wie die Progammdateien benannt werden sollen, und welche Funktionalität in welchen Klassen und Methoden implementiert werden soll. Trotzdem überprüft UniWorX zur Zeit weder, ob alle Aufgaben des Übungsblattes abgegeben wurden, noch ob die Struktur der Abgaben den Vorgaben entspricht. Vergisst ein Student also beispielsweise eine Aufgabe hochzuladen, oder benennt er ein Element anders als in der Aufgabenstellung gefordert, wird dies erst dem Korrektor auffallen. Konkret bedeutet dies für den Studenten, dass üblicherweise eine Korrektur ausbleibt, und dass ihm Punkte für die Übung nicht anerkannt werden können. Diese Situation ist für beide Seiten unbefriedigend.

Entspricht die Programmstruktur nicht der Aufgabenstellung, erschwert dies die Arbeit des Korrektors. Zudem ist es schwierig zu beurteilen, ob ein gegebenes Programm die gewünschte Funktionalität auch korrekt implementiert. Schließlich steht zur Korrektur nur begrenzt viel Zeit zur Verfügung, und normalerweise wird ein Programm nur dann ausgeführt, wenn dies unbedingt erforderlich scheint. Trotzdem kann beispielsweise die korrekte Behandlung von allen Rand- und Sonderfällen nur selten im Detail überprüft werden. Darüber hinaus bleibt wenig Zeit um zusätzliche Qualitätsmerkmale, wie den Programmierstil, zu kommentieren, oder weiterführende Hinweise zu geben, obwohl dies gerade für angehende Informatiker mit wenig Programmiererfahrung besonders hilfreich wäre.

1.2 Automatisiertes Testen

Ein Grundparadigma der Workflow-Unterstützung ist die weitgehende Automatisierung von zeitaufwendigen Aufgaben. Im Fall der Abgabenkorrektur ist dies das Testen der Programme. Im Folgenden soll daher untersucht werden, wie dies sinnvoll automatisiert werden kann.

Ein Programm besitzt zahlreiche strukturelle Eigenschaften: Die äußere Struktur lässt sich allgemein durch die Dateistruktur spezifizieren. Die innere Struktur hingegen kann je nach Art der Programmiersprache aus unterschiedlichen Elementen bestehen. Ob die äußere Struktur der Aufgabenstellung entspricht, lässt sich einfach überprüfen. Um darüber hinaus die innere Struktur herauszufinden und zu überprüfen, gibt es grundsätzlich zwei Möglichkeiten: Die erste Möglichkeit ist es, das Programm auszuführen und mittels Reflection zur Laufzeit die Struktur auszulesen. Der Vorteil davon ist, dass dafür in fast allen Programmiersprachen Bibliotheken verfügbar sind, wie z.B. die Reflection API [FF04] in Java. Der Nachteil davon ist, dass durch das Ausführen des Programms ein grundsätzliches Sicherheitsproblem entsteht. Deshalb fiel hier die Entscheidung auf die zweite Möglichkeit, die Programmmerkmale durch Parsen der Programmsyntax herauszufinden. Dieser Vorgang ist an sich sicher, sofern man voraussetzen kann, dass der verwendete Parser selbst keine Exploits besitzt, da das Programm nicht ausgeführt werden muss.

Darüber hinaus gilt es auch die funktionalen Eigenschaften eines Programmes zu überprüfen: Erfüllt das Programm alle funktionalen Anforderungen der Aufgabenstellung und werden insbesondere alle Sonderfälle richtig behandelt? Hier helfen sogenannte Unit-Tests, die genügend Testszenarien enthalten, um das korrekte Programmverhalten sicherzustellen. Hierzu muss man das gegebene Programm allerdings tatsächlich ausführen. Dies darf daher nur in einer sicheren Sandbox geschehen, um zu verhindern, dass fehlerhafter oder schadhafter Code das ausführende System beeinträchtigt. In Java wird einem das durch das implementierte Sicherheitsmodell [GED03] ermöglicht, das man komfortabel konfigurieren kann.

1.3 Umsetzung

Den gewählten Techniken zum Testen ist gemein, dass sie sich automatisiert ausführen lassen. Die Grundvoraussetzung für eine softwaretechnische Umsetzung ist also erfüllt. Bleibt also die Frage, wie und an welcher Stelle man dies alles in den bestehenden Abgabeprozess integrieren kann. Wenn der Prozess wie beschrieben verbessert werden soll, muss hierzu schon bei der Stellung der Aufgaben begonnen werden: Die Vorgaben hinsichtlich Struktur und Funktionalität wie bisher nur textuell in der Aufgabenstellung zu beschreiben, reicht dafür natürlich nicht mehr aus. Diese Vorgaben müssen daher von Anfang an spezifiziert werden, und beim Erstellen eines Übungsblattes im Abgabesystem mit hinterlegt werden. Gibt ein Student nun ab, so kann die Abgabe auf ihre Strukturmerkmale hin getestet, und dem Studenten unmittelbar darauf das Ergebnis als Feedback präsentiert werden. Zwischen dem Zeitpunkt der Abgabe und dem Zeitpunkt der Korrektur verbleibt genug Zeit, um durch UniWorX nicht nur die strukturellen, sondern auch die funktionalen Tests durchzuführen. Diese vollständigen Testergebnisse können dann den Korrektoren zur Verfügung gestellt werden, was diesen die Korrektur erheblich erleichtert.

Die besonderen Herausforderungen hierbei sind folgende:

1. Dem Aufgabensteller einen geeigneten Editor zur Verfügung zu stellen, in dem er die Anforderungen an das Programm spezifizieren kann.
2. Eine Infrastruktur zum sicheren Testen der abgegebenen Programme zu konzipieren.
3. Beides nahtlos in das bestehende Abgabesystem UniWorX so zu integrieren, dass sowohl Studenten als auch Korrektoren davon größtmöglich profitieren können.

Die nun folgende Darstellung der Umsetzung des genannten Vorhabens gliedert sich wie folgt: Zuerst wird die grundsätzlichen Methodik erläutert, die für die Umsetzung verwendet wurde. Danach werden die Anforderungen im Detail ausgearbeitet, bevor das daraus resultierende Design diskutiert wird, welches nicht nur beschrieben, sondern auch anhand eines Beispielprozess evaluiert wird. Schließlich werden dann die Ergebnisse präsentiert, an welche sich noch ein Ausblick für künftige Erweiterungsmöglichkeiten, sowie ein abschließendes Fazit anknüpft.

2 Methodik

Ein essentieller Punkt für jede erfolgreiche Entwicklung ist die Methodik, nach welcher der gesamte Prozess von der Entwicklung bis zur Planung durchgeführt wird. Denn eine falsche Wahl kann nicht nur die Umsetzung verzögern, sondern sogar manchmal gar die Fertigstellung als solches gefährden. Speziell bei der Entwicklung von UniBene hat die richtige Wahl der Methodik nicht unerheblich zum Erfolg und zur zügigen Fertigstellung des Projektes beigetragen, weswegen im Folgenden nicht nur erläutert werden soll, welche Methodik gewählt wurde, sondern auch, welche Kriterien für die Wahl ausschlaggebend waren.

2.1 Entwicklungsprozess

Ein wichtiger Teil der Methodik ist das gewählte Vorgehen für den Entwicklungsprozess. Die klaren Anforderungen würden natürlich vermuten lassen, dass ein sequentielles Vorgehen hier besonders geeignet wäre. In der Tat hat man, wenn man sequentiell vorgeht, einen geringen Planungsaufwand, solange die Anforderungen konstant bleiben. Allerdings geht aus den Anforderungen auch hervor, dass mehrere unabhängige Komponenten zur Realisierung erforderlich sind. Deren Integration in und Kommunikation mit UniWorX sind natürlich eine integrale Notwendigkeit, deren Komplexität nicht zu unterschätzen ist. Die mit dem sequentiellen Vorgehen einhergehende „Big Bang"-Integration wäre also denkbar ungünstig gewesen.

Nicht zuletzt schien daher ein iteratives Vorgehen von Vorteil, welches einem eine dem Problem angemessenere inkrementelle Entwicklung erlaubt. Dadurch können die einzelnen Komponenten schon früh miteinander integriert werden, was die Komplexität der Integration deutlich verringert. Ein weiterer Vorteil ist, dass sich die Iterationen gut dafür eignen, die Funktionalität nach Abhängigkeiten und Prioritäten auf diese zu verteilen. Typischerweise wandern so weniger wichtige Funktionen in spätere Iterationen, so dass man sich zu Anfang auf die Kernfunktionalität konzentrieren kann, und den Rest dann später immer noch neu auf seine Notwendigkeit hin evaluieren kann. Bei manchen anfangs zwar spezifizierten, aber als nicht ganz so wichtig eingestuften Funktionen stellt sich dann später auch heraus, dass diese eigentlich gar nicht mehr notwendig sind, wodurch der sonst möglicherweise investierte Aufwand für diese Funktionen eingespart wird. Eine Herausforderung bei so einem iterativen Vorgehen ist allerdings die im Vergleich zum sequentiellen Vorgehen deutlich komplexere Projektplanung, die daher im besonderen Maße sorgfältig durchgeführt werden sollte, um den Überblick nicht zu verlieren.

2.2 Projektplanung

Die schon erwähnte funktionale Grobstrukturierung zur Aufteilung auf die verschiedenen Iterationen diente als Grundlage für die Projektplanung. Konkret ergaben sich hierbei folgende drei Iterationen, von denen jede so konzipiert ist, dass man nach jeder ein lauffähiges Release der Software erhält:

1. Grundlegendes Release
2. Fortgeschrittenes Release
3. Bonus-Release

Während für das grundlegende Release der Schwerpunkt primär auf der Infrastruktur und ersten lauffähigen Versionen aller Komponenten lag, waren für das fortgeschrittene Release, dann schon die ganze Palette an strukturellen und funktionalen Tests eingeplant. Im Bonus-Release schließlich waren alle weiteren Funktionen enthalten, die für die Kernfunktionalität nicht zwingend notwendig waren. Mit der Zielsetzung mindestens das fortgeschrittene Release fertigzustellen, war das grobe Zeitraster festgelegt.

Für die Planung der einzelnen Iterationen habe ich ausgehend von einem Grobdesign eine funktionale Zergliederung vorgenommen. Darauf aufbauend konnte ich nun den Aufwand für die einzelnen Funktionsbestandteile schätzen. Allerdings war es damit nicht getan, denn um das Projekt vernünftig steuern zu können, musste ich natürlich auch die tatsächlichen Aufwände erfassen, um darauf basierend dann die Schätzung für die gesamte Iteration korrigieren zu können. Um dies realisieren zu können, entschied ich mich aus pragmatischen Gründen für die Umsetzung mit einer Tabellenkalkulation, um den Aufwand möglichst gering zu halten.

15	**# Programmstruktur (Klassen, Methoden)**	
16	# Im Editor erfassen und abspeichern	**4**
17	# Importieren	**4**
18	# Testen dieser im Server (Schnittstellen ...)	**8**

Abbildung 2.1: Funktionale Zergliederung

In Abbildung 2.1 kann man einen Teil der funktionalen Zergliederung sehen: Neben jedem konkreten Funktionsbestandteil ist der geschätzte Stundenaufwand notiert.

	67,5	**52,5**	**52,0**
	Invested	**Burned**	**Earned**

Abbildung 2.2: Zusammengefasste Stundenanzahl

Abbildung 2.2 zeigt die zusammengefasste Stundenanzahl, aufgeteilt in drei Kategorien: Invested sind die Stunden, die insgesamt schon aufgewendet wurden. Burned analog, aber nur unter Berücksichtigung der schon fertigen Funktionsbestandteile. Earned bezieht sich ebenfalls nur auf die fertigen Funktionsbestandteile und summiert deren anfänglich geschätzten Aufwände.

Ausgehend von den genannten Werten Invested, Burned und Earned kann man direkt die in Abbildung 2.3 ersichtlichen Kennzahlen SPI und CPI berechnen.

39	Gesamtaufwand in Tagen	28,0	40,7	28,2
40	Restaufwand in Tagen	16,5	29,2	16,7
41	Fertigstellungsdatum	27.08.08	09.09.08	27.08.08
42	Zusatzaufwand in Tagen	5,0	17,7	5,2
43			SPI (> 1)	CPI (> 1)
44		Kennzahl	0,57	0,99

Abbildung 2.3: Kennzahlen mit Restaufwandsschätzung

SPI steht für „Schedule Performance Indicator“, und berechnet sich aus dem Verhältnis von Earned und dem, was man gemäß Plan schon geschafft haben sollte: Man kann daraus also ablesen, wie gut man noch im Zeitplan liegt. CPI steht im Gegensatz dazu für „Cost Performance Indicator“, und berechnet sich aus dem Verhältnis von Earned und Burned. Daraus kann man ablesen, wie richtig man mit der anfänglichen Aufwandsschätzung lag. Nun kann man mit diesen Kennzahlen jeweils eine korrigierte Schätzung für den Restaufwand berechnen, die ebenfalls in der Abbildung zu sehen ist.

So hat man also ein Werkzeug an der Hand, mit dem man jederzeit den Projektfortschritt messen kann, und anhand dessen man künftige Planungen, beispielsweise für folgende Iterationen, weiter verbessern kann. Wie effektiv die Projektplanung in der Praxis tatsächlich war, darauf soll später bei den Ergebnissen eingegangen werden, denn um dies betrachten zu können, sind sowohl die Anforderungen als auch das daraus resultierende Design von Belang, welche beide im Folgenden betrachtet werden sollen.

3 Anforderungen

Nun also zu den konkreten Anforderungen an das zu entwickelnde System. Schauen wir uns dazu aber zuerst die übergeordneten Ziele an, die überhaupt erreicht werden sollen.

3.1 Ziele

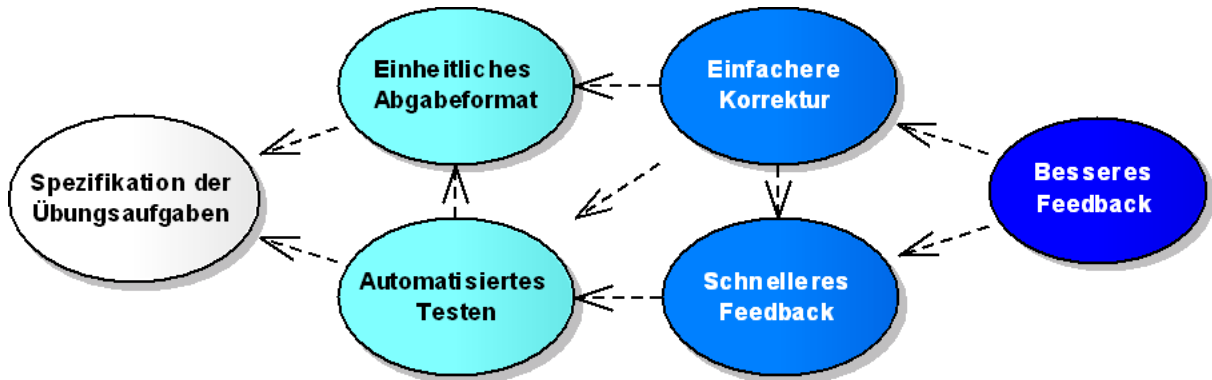


Abbildung 3.1: Zielabhängigkeiten

Wie man in Abbildung 3.1 sieht, ist die Spezifikation der Übungsaufgaben die Grundlage für alle weiteren Ziele. In dieser muss genau beschrieben werden, wie ein abzugebendes Programm strukturiert werden soll, und welche Funktionalität es erfüllen soll. Damit kann ein einheitliches Abgabeformat sichergestellt werden, wodurch ein automatisiertes Testen überhaupt erst ermöglicht wird.

Das automatisierte Testen wiederum hat zwei grundlegende Folgen: Einmal kann durch die Automatisierung ein (neues) schnelleres Feedback bereits direkt nach der Abgabe realisiert werden, das den Studenten schon auf strukturelle Mängel aufmerksam machen kann. Des Weiteren tragen die strukturellen und funktionalen Tests auch zu einer einfacheren Korrektur bei, da man dies nun nicht mehr manuell überprüfen muss. Das einheitliche Abgabeformat trägt natürlich ebenfalls zu einer einfacheren Korrektur bei, da nicht jede Lösung allein strukturell schon komplett unterschiedlich ist. Auch das bereits erwähnte schnellere Feedback trägt zur Vereinfachung der Korrektur bei, denn dadurch können die Studenten schon vor der Korrektur einfache Fehler ausbessern, so dass man hochwertigere Abgaben erhält.

Und schließlich tragen sowohl die einfachere Korrektur, die dem Korrektor mehr Zeit für weiterführende Kommentare lässt, als auch das schnellere Feedback, das die Studenten zu besseren Abgaben ermutigt, beide zu dem Gesamtziel bei, das Feedback für die Studenten insgesamt deutlich zu verbessern.

3.2 Abgabeprozess

Nachdem wir nun die übergeordneten Ziele gesehen haben, wird es Zeit sich die Anforderungen im Detail anzuschauen. Verschaffen wir uns dazu also zuerst einen Überblick über den für uns relevanten Teil des Abgabeprozesses im Kontext von UniWorX und UniBene.

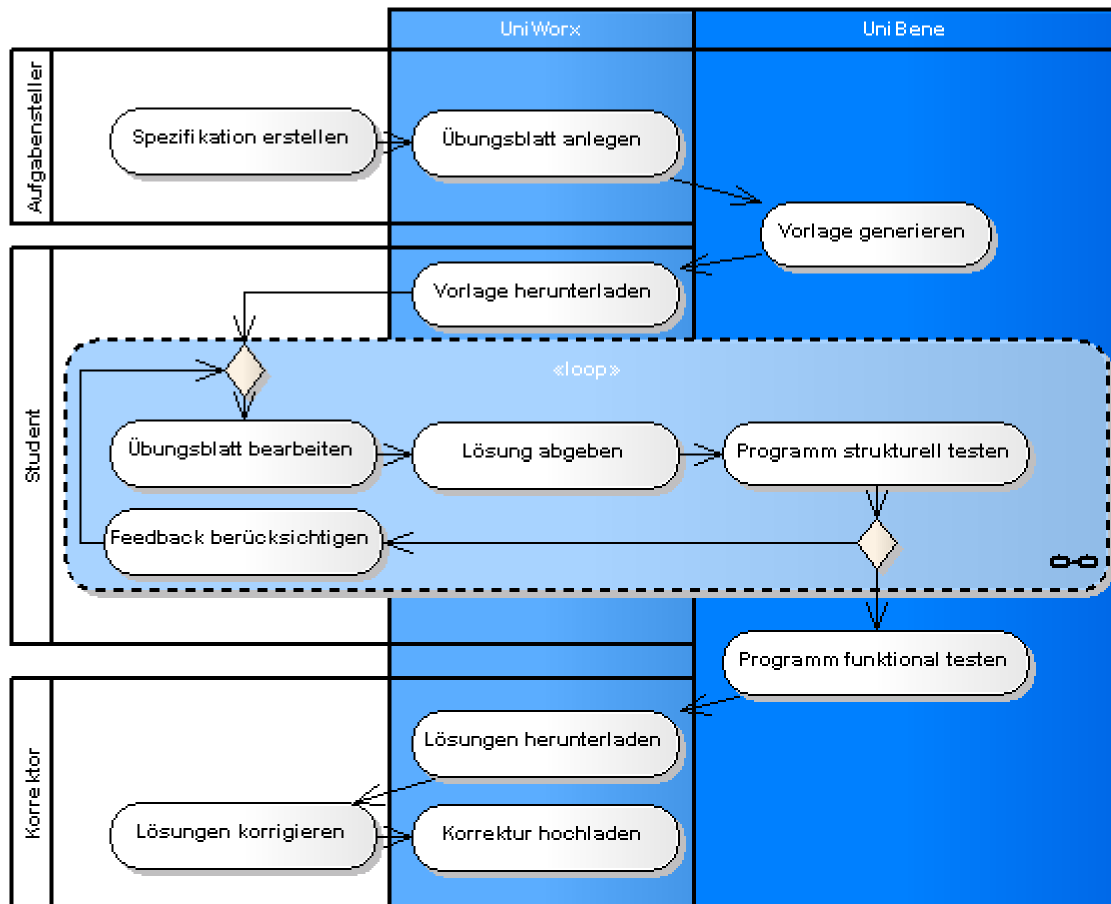


Abbildung 3.2: Überblick über den Abgabeprozess

Der Workflow (Abbildung 3.2) beginnt damit, dass der Aufgabensteller die Spezifikation des Übungsblattes erstellt, und in UniWorX ein dazugehöriges Übungsblatt anlegt.

Ab diesem Zeitpunkt kann der Student die von UniBene generierte Vorlage herunterladen, und bis zur Abgabefrist beliebig oft eine Lösung abgeben. Bei der Abgabe wird die Abgabe strukturell geprüft, und das Feedback daraus dem Studenten präsentiert.

Sobald die Abgabefrist abgelaufen ist, werden die Abgaben durch UniBene auch funktional getestet, und mit den Testergebnissen versehen an die Korrektoren verteilt. Schicken die Korrektoren nun ihrerseits die korrigierten Abgaben wieder an UniWorX zurück, trägt UniWorX dafür Sorge, dass den Studenten ihre Korrektur zugestellt wird.

Der beschriebene Abgabeprozess kann nun als Orientierung dienen, um die einzelnen Funktionen, die benötigt werden, gemäß ihrer Reihenfolge in selbigem im Detail zu betrachten.

3.3 Funktionalität

Im Folgenden sollen daher zuerst die Kernfunktionen im Rahmen des Abgabeprozesses beschrieben werden, bevor noch näher auf die Integration mit UniWorX, sowie die technischen Anforderungen eingegangen werden soll.

3.3.1 Spezifikation

Die erwähnte Spezifikation eines Übungsblattes wird durch den Aufgabensteller folgendermaßen durchgeführt: Zuerst einmal muss die Grundstruktur eines Übungsblattes erfasst werden, sprich man muss einzelne Aufgaben anlegen, sowie diese mit Teilaufgaben versehen können. Zu jeder Aufgabe bzw. Teilaufgabe muss definiert werden können, welche Dateien und Verzeichnisse erwartet werden. Noch genauer sollte man erwartete Programmdateien spezifizieren können: Erwartete Klassen und Methoden, aber auch benötigte Sicherheitsprivilegien und Unit-Tests sollten angegeben werden können. Des Weiteren sollte man diese Sachen nicht nur manuell angeben können, sondern, soweit möglich, auch aus einer bestehenden Musterlösung importieren können, die man normaler Weise sowieso schon zu jeder Aufgabe erstellen muss. Schließlich muss der Zusatzaufwand für den Aufgabensteller möglichst gering gehalten werden, um eine Akzeptanz des Systems nicht zu gefährden. Zudem wären verschiedene Exportfunktionen wünschenswert: Hat man beispielsweise schon die Dateistruktur der Abgabe spezifiziert, sollte es nicht allzu schwer sein, für die Studenten eine leere Vorlage zu erstellen, um diesen die korrekte Abgabe zu erleichtern. Auch wäre ein Export in diverse Textformate zur Verwendung in dem Aufgabentext denkbar, da dieser ja auch die gewünschten Vorgaben für die Abgabe erläutern muss. Eine standardisierte Schreibweise könnte hierbei sowohl dem Aufgabensteller Arbeit abnehmen, als auch den Studenten das Verständnis erleichtern.

3.3.2 Abgabe

Wurde eine Spezifikation für ein Übungsblatt erstellt, muss es möglich sein, bei der Anlage eine Übungsblattes in UniWorX diese Spezifikation mit anzugeben. Dadurch könnte den Studenten dann schon vor der Abgabe die Möglichkeit gegeben werden, sich die geforderte Struktur der Abgabe anzusehen, und die schon erwähnte Vorlage herunterzuladen. Bei der Abgabe kann UniWorX nun unterscheiden, ob zu dem betreffenden Übungsblatt eine Spezifikation vorhanden ist: Falls ja, sollte dem Studenten darüber Feedback gegeben werden, inwiefern er die strukturellen Vorgaben erfüllt hat, damit er seine Abgabe nochmals nachbessern kann. Dazu muss natürlich eine entsprechende Validierungskomponente, welche eine Abgabe auf ihre Struktur hin überprüfen kann, in UniWorX integriert werden. Da das Feedback unmittelbar erfolgen sollte, und es schließlich durchaus vorkommen kann, dass zu bestimmten Zeiten sehr viele Studenten auf einmal abgeben, ist es nicht unbedingt sinnvoll auch gleich eine funktionale Validierung durchzuführen. Aus Sicherheitsgründen ist es zudem sinnvoll, die abgegebenen Programme nicht direkt in UniWorX auszuführen, um sie funktional zu validieren.

3.3.3 Korrektur

Ist nun die Abgabefrist eines Übungsblattes verstrichen, so sollten nicht wie bisher sofort die Abgaben an die Korrektoren verteilt werden. Vielmehr sollten vorher die ebenfalls spezifizierten funktionalen Tests auf den Abgaben ausgeführt werden, damit deren Ergebnisse den Korrektoren zusammen mit den Abgaben übergeben werden können. Typischerweise können solche Funktionstests auch einmal etwas länger dauern, insbesondere bei einer größeren Anzahl an Abgaben. Auch müssen manchmal Programme getestet werden, die limitierten Zugriff auf das Dateisystem oder auf das Netzwerk benötigen. Beides sollte UniWorX selber weder in Bezug auf Performance noch auf Sicherheit beeinträchtigen können, da die ständige und zuverlässige Verfügbarkeit von UniWorX gewährleistet werden muss, damit Studenten jederzeit ihre Lösungen abgeben können. Hierbei muss auch berücksichtigt werden, dass möglicherweise die Abgabefristen mehrerer Übungsblätter verschiedener Übungen zeitlich dicht zusammenfallen können. Dies sollte ebenfalls keinerlei bemerkbare Auswirkung auf UniWorX haben, da es durchaus vorkommen kann, dass viele Studenten erst kurz vor der Abgabefrist abgeben, so dass ein Ausfall zu so einem Zeitpunkt als kritisch anzusehen ist.

3.3.4 Integration mit UniWorX

Über die genannten Kernfunktionen hinaus gibt es noch weitere Anforderungen. Eine grundlegende ist insbesondere die Integration mit UniWorX: Auf der einen Seite müssen die Spezifikationen für die Übungsblätter in UniWorX eingepflegt werden, damit aus diesen bereits direkt bei jeder Abgabe ein erstes Feedback generiert werden kann. Hier muss also darauf geachtet werden, dass eine nahtlose Integration gelingt, und kein Mehraufwand für den Aufgabensteller entsteht. Trotzdem sollte die Integration aber so unaufdringlich bleiben, dass man immer noch Übungsblätter ohne dazugehörige Spezifikation anlegen kann, und dass man eine Lösung trotzdem abgeben kann, auch wenn ihre Struktur nicht exakt der spezifizierten Struktur entspricht. Schließlich will man ja vermeiden, dass die Studenten plötzlich gar nicht mehr abgeben können. Auf der anderen Seite muss darauf geachtet werden, wie viel Funktionalität direkt in UniWorX integriert wird. Denn die direkte Einbindung der Funktionstests kann potientielle Performanceprobleme und Sicherheitsprobleme nach sich ziehen. Eine Auslagerung dieser Funktionalität ist also zu erwägen, wobei man dann dafür im Gegenzug für die Kommunikation mit UniWorX genaue Schnittstellen definieren muss.

3.3.5 Sicherheit

Dies bringt uns zu der schon mehrmals angesprochenen Sicherheitsproblematik: Möchte man Programme auf ihre Funktionalität hin testen, so muss man sie dafür auch tatsächlich ausführen. Da man bei den Abgaben weder Programmierfehler noch unzulässige Aufrufe ausschließen kann, muss man diese konsequenterweise als Fremdcode betrachten, und sie daher in einer sicheren Sandbox ausführen. Zu beachten ist dabei, dass manchmal die Aufgabenstellung auch intrinsisch sicherheitskritische Aktionen, wie zum Beispiel die Erstellung von Dateien, erfordern kann. Man kann also bei der Ausführung nicht einfach pauschal alle sicherheitskritischen Aktionen verbieten, sondern sollte stattdessen ermöglichen, in der Spezifikation erlaubte Aktionen genau angeben zu können. Zu weitreichende Rechte sollte man allerdings auch nicht spezifizieren können, da die Spezifikation ja von Übungsleitern erstellt wird, also von Benutzern des Systems. Hier muss man also Kompromisse eingehen, denn die funktionale Validierung darf unter keinen Umständen die Integrität des Servers kompromittieren.

3.3.6 Sprachunabhängigkeit

Eine weitere Herausforderung, die bisher noch nicht erwähnt wurde, ist die Sprachunabhängigkeit. Ist doch UniWorX in Java geschrieben, liegt es auch nahe UniBene mit Java zu realisieren. Doch was für UniBene an sich gilt, muss nicht zwangsläufig auch für die potentiellen Abgaben gelten. Zwar werden in der Tat viele Programmieraufgaben für Java gestellt, allerdings ist es durchaus auch üblich funktionale Programmiersprachen wie SML einzusetzen, oder für spezielle Zwecke andere besonders geeignete Sprachen zu verwenden. Diesem Umstand sollte man am Besten schon beim Design Rechnung tragen, um die Einbindung neuer Programmiersprachen prinzipiellzu ermöglichen. Schließlich dürfte es schwer fallen, dies nachträglich zu berücksichtigen, wenn die notwendige Flexibilität dahingehend nicht von Anfang an eingeplant wird.

4 Design und Evaluation

Nachdem wir nun die Anforderungen gesehen haben, soll das daraus für UniBene resultierende Design vorgestellt werden. Ausgehend von einem Überblick über die Gesamtstruktur, soll schließlich auf die enthaltenen Komponenten im Detail eingegangen werden. Damit einhergehend findet eine Evaluation anhand eines durchgehenden Beispielprozesses statt.

4.1 Überblick

Dieser Prozess wird in Abbildung 4.1 im Kontext der Gesamtstruktur von UniBene und UniWorX beschrieben, wobei die Artefakte gemäß der zeitlichen Abfolge nummeriert sind.

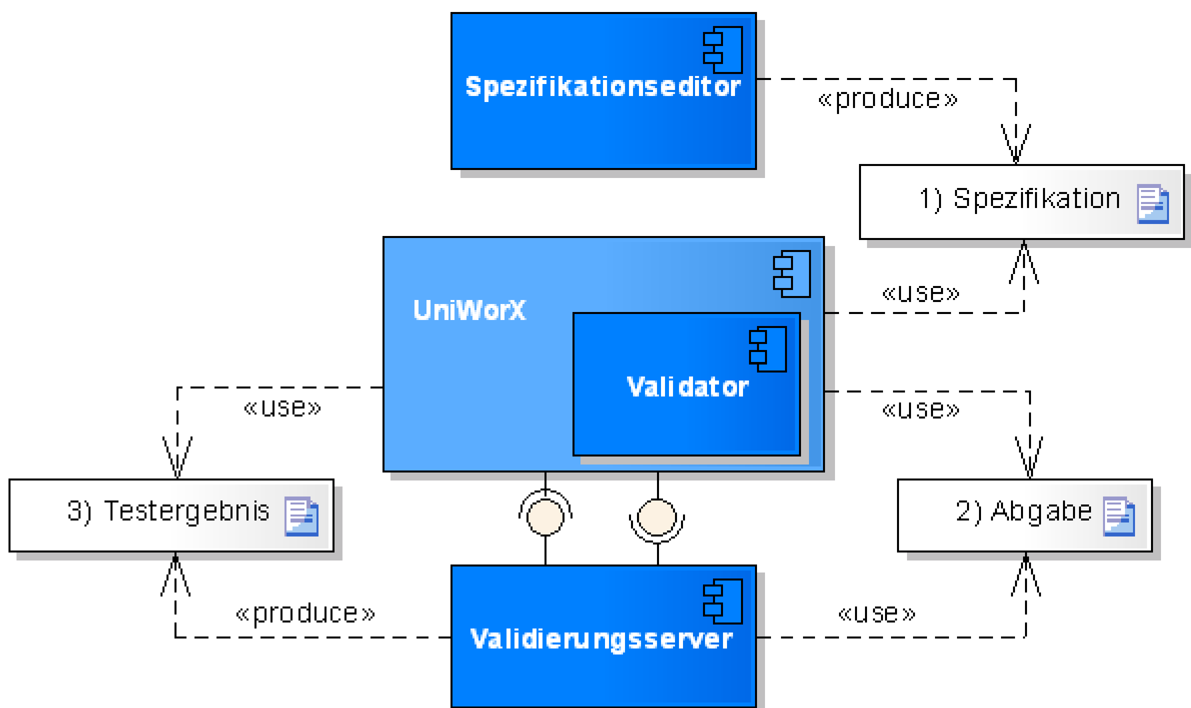


Abbildung 4.1: Gesamtstruktur

Die Gesamtstruktur (Abbildung 4.1) besteht aus dem schon vorhandenen UniWorX, sowie der Erweiterung von UniWorX mit UniBene um die folgenden drei Komponenten:

1. Im Spezifikationseditor spezifiziert der Aufgabensteller seine Übungsaufgaben.
2. Über die Validator-Komponente kann UniWorX hochgeladene Spezifikationen nutzen, um Vorlagen für die Studenten zu generieren, und um die Abgaben auf ihre Struktur hin zu validieren, sowie die Ergebnisse hieraus den Studenten als Feedback zu präsentieren.
3. Im Validierungsserver werden die über UniWorX bezogenen Abgaben mit Unit-Tests auf ihre funktionale Korrektheit hin geprüft, und die Testergebnisse daraus UniWorX zur Weitergabe an die Korrekturen zur Verfügung gestellt.

4.2 Spezifikationseditor

Der Spezifikationseditor ist eine Java-Anwendung, die es dem Aufgabensteller erlaubt, ein Übungsblatt genau zu spezifizieren. Er kann einzelne Aufgaben und Teilaufgaben definieren, Verzeichnisse und Dateien anlegen, und speziell für Java Pakete, Klassen, Methoden, etc. benennen, sowie auch Unit-Tests bereitstellen, die die Einhaltung der Funktionalität überprüfen, und deren erforderliche Sicherheitsprivilegien festgelegt werden können.

4.2.1 YAML

Zum Speichern solcher Spezifikationen wird YAML [BKEdN08] verwendet, ein programmiersprachenunabhängiges Serialisierungsformat. Das folgende einfache Beispiel illustriert wie ein Mapping mit einer Sequenz darin in YAML aussieht:

```
1 foo: "hello world"
2 bar:
3  - hello
4  - world
```

Die Entscheidung für YAML fiel aus mehreren Gründen:

1. Baumstruktur: Eine Spezifikation kann man sich intuitiv als Baum vorstellen, denn Übungsblätter enthalten Aufgaben, Aufgaben enthalten Teilaufgaben, Aufgaben und Teilaufgaben enthalten Verzeichnisse und Dateien, etc.

2. Erweiterbar: Nicht zuletzt auch wegen der erstrebten Sprachunabhängigkeit und wegen der vielfältigen Validierungsmöglichkeiten und -optionen muss sich das Format leicht erweitern lassen.

3. Leichtgewichtig: Spezifikationen müssen an vielen Stellen in UniWorX und im Validierungsserver gelesen werden, eine minimale Syntax, die ein schnelles Parsen ermöglicht, ist hier von Vorteil.

4. Menschenlesbar: Auch ohne graphischen Editor kann man so relativ einfach von Hand einzelne Dateien selbst schreiben oder generierte Dateien überprüfen.

In YAML werden allerdings nicht nur die Spezifikationen gespeichert, sondern auch deren Metamodell, so dass auch hier der Erweiterbarkeit Rechnung getragen wird. Für dieses Metamodell gibt es zwar keinen Editor, aber da YAML ja als menschenlesbares Format konzipiert wurde, ist dies auch so gut editierbar.

Um aber eine einfache Erweiterbarkeit wirklich konsequent zu gewährleisten, muss darüber hinaus auch der Editor, genauer dessen graphische Benutzeroberfläche leicht erweiterbar sein. Aus diesem Grund ist selbige nicht hardcodiert, sondern wird aus erwähntem Metamodell generiert, das dazu annotiert wird.

4.2.2 Funktionen

Wie selbige Benutzeroberfläche für die Spezifikation eines einfach aufgebauten Übungsblattes aussieht, kann man in Abbildung 4.2 sehen:

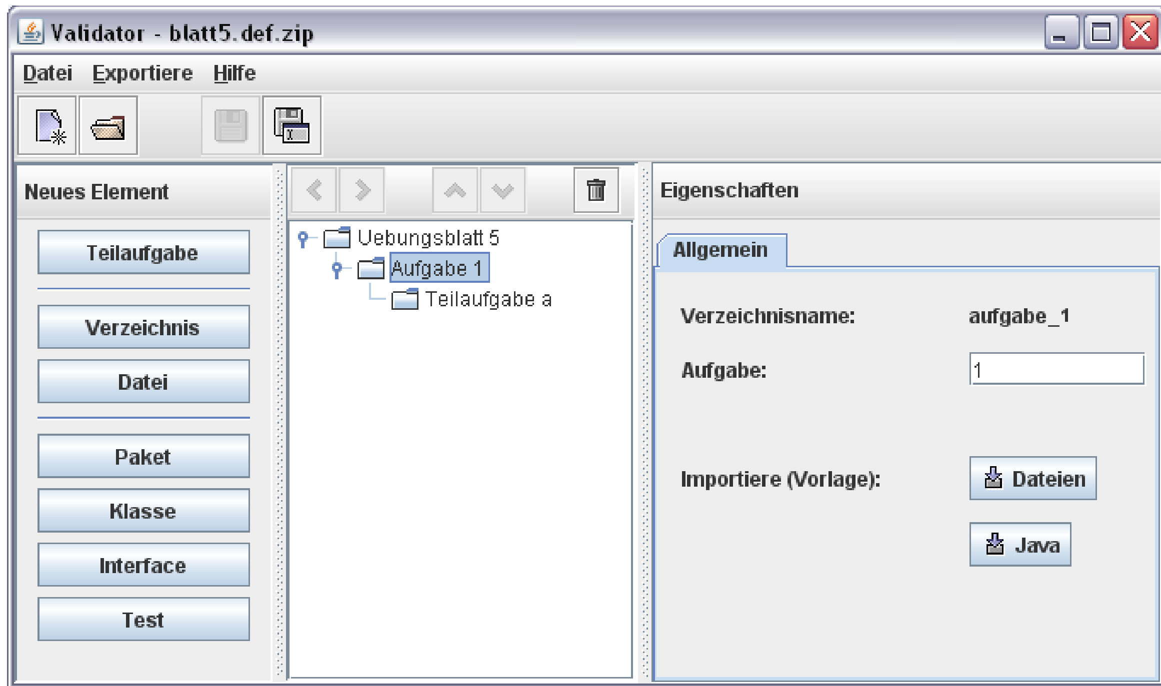
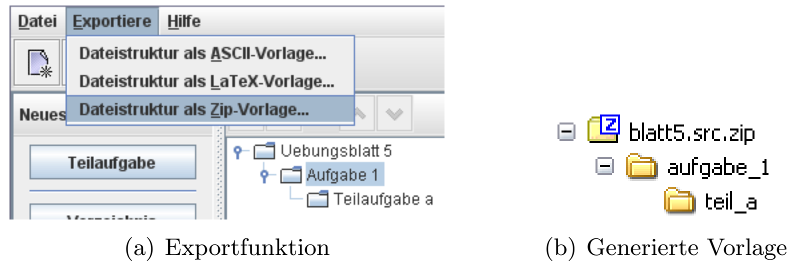


Abbildung 4.2: Einfach aufgebautes Übungsblatt im Editor

Das Wurzelelement der Spezifikation ist das Übungsblatt selbst. Ein Übungsblatt kann man in Aufgaben, und die Aufgaben wiederum in Teilaufgaben unterteilen. Gemäß Metamodell erlaubte Kindelemente werden hierbei links unter „Neues Element" gelistet und können dort unter dem ausgewählten Element der Spezifikation erstellt werden. Die Eigenschaften des ausgewählten Elements können rechts editiert werden. Da die Aufgabenstruktur eines Übungsblattes üblicherweise seine Entsprechung in der Verzeichnisstruktur der Abgabe findet, leitet der Editor diese implizit ab, und zeigt den abgeleiteten Verzeichnisnamen im entsprechenden Feld rechts bei den Eigenschaften an.

Exportiert man eine Vorlage für die Studenten, so wird eine entsprechende Verzeichnisstruktur generiert, wie in Abbildung 4.3 zu sehen ist.



(a) Exportfunktion (b) Generierte Vorlage

Abbildung 4.3: Abgeleitete Verzeichnisstruktur

Eine explizite Verzeichnisstruktur kann man natürlich zusätzlich definieren. Auch bestimmte Dateien können spezifiziert werden, wobei man hier auch die Möglichkeit hat, für diese eine Dateivorlage anzugeben, welche dann in erwähntem Export für die Studenten entsprechend hinzugefügt wird.

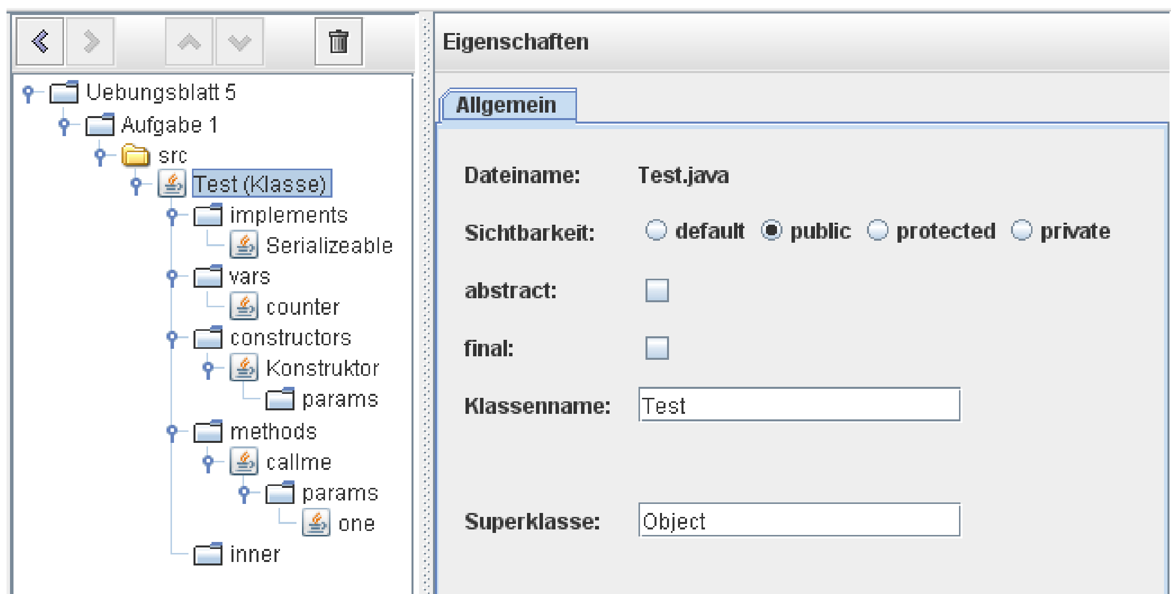


Abbildung 4.4: Java-Klasse im Editor

Darüber hinaus kann man auch die geforderte Struktur von Programmen spezifizieren: In Abbildung 4.4 ist beispielsweise die Klasse Test, samt Konstruktor, Variablen und Methoden definiert, wobei die Klasse in einem src-Unterverzeichnis der Aufgabe 1 abgeben werden soll.

Oft wird man ja zu jeder Aufgabe (Teil-)Vorlagen erstellen, sowie eine Musterlösung anlegen, um selbst zu sehen, wie die Aufgabe zu lösen ist. Deshalb kann man die Aufgabenstruktur, die Datei- und Verzeichnisstruktur, und die Programmstruktur auch aus einem vorhandenen Verzeichnis importieren, sofern man sich an die verwendeten Namenskonventionen hält.

4.3 UniWorX-Integration

Hat der Aufgabensteller nun die Spezifikation für ein Übungsblatt mit dem Editor erstellt, so kann er diese bei der Erstellung des Übungsblattes in UniWorX angeben:

Übungsblatt anlegen/bearbeiten (2 von 3)

Vorlesung: InfoSys - 123

Blattnummer: Blatt 11
Bewertungstyp: Punkte-Bewertung
Vorgabe [optional] : Durchsuchen...
Gewünschtes Dateiformat: .def.zip
Maximale Dateigröße: 10 MB
Deadline: 15.09.2008 16:00 Format: dd.MM.yyyy HH:mm
Bonusblatt:
Aktuelles Blatt:
Anlegen

Abbildung 4.5: Übungsblatt anlegen

Ohne Spezifikation verfährt UniWorX genau wie bisher, wird aber eine Spezifikation angegeben, so macht sich dies an mehreren Stellen bereits vor einer ersten Abgabe bemerkbar:

Vorgabe
Übungsblatt: Blatt 11
Dateistruktur: Abgaben sollten folgendermaßen aufgebaut sein:

- aufgabe_1
  - teil_a
    - source
      - Time.java
    - test.gif
  - teil_b
- aufgabe_2
  - TimeStopper.jar

Folgendes Zip-Archiv enthält eine Vorlage für obige Dateistruktur:

- blatt11.src.zip
  - Während die leeren Dateien darin lediglich zum Überschreiben vorgesehen sind,
  - können die anderen Dateien direkt als Vorlage bzw. Hilfestellung verwendet werden.

Abbildung 4.6: Spezifikationsinformationen

Sowohl Aufgabensteller als auch die Studenten können sich die in der Spezifikation definierte Struktur ansehen, und die bereits vorher erwähnte Zip-Vorlage herunterladen - in Abbildung 4.6 sehen wir die Studentensicht auf die Vorlage. Der Aufgabensteller hat in seiner Ansicht noch zusätzlich die Möglichkeit, die Spezifikation selbst herunterzuladen.

Gibt ein Student nun eine Lösung für das angelegte Übungsblatt ab, so kann er sich erwähnte Informationen über die dazugehörige Spezifikation vorher anzeigen lassen. Nach seiner Abgabe erhält er dann unmittelbar ein detailliertes Feedback, inwieweit seine Abgabe strukturell konform mit der Spezifikation ist.

Lösung abgeben (2 von 2)

Danke, Du hast gerade erfolgreich Deine Lösung abgegeben

Die Dateistruktur deiner Abgabe wurde folgendermaßen gegen die Vorgabe verglichen:

- **Abgabe**
  - **blatt11.zip [erwartet]**
    - **aufgabe_1 [erwartet]**
      - **blue.png [unerwartet]**
      - **teil_a [fehlt]**
      - **teil_b [fehlt]**
    - **aufgabe_2 [erwartet]**
      - **TimeStopper.jar [leer]**
    - **text.txt [unerwartet]**

Legende:

- **[erwartet]**
  - **Abgegeben, und auch gemäß der Vorgabe so erwartet:**
    - Du musst nichts weiter tun.
- **[leer]**
  - **Mit leerem Inhalt abgegeben:**
    - Diese Dateien hätten eigentlich nicht leer sein sollen.
    - Falls dies nicht deine Absicht war, solltest du kontrollieren,
      - ob du diese Dateien vergessen hast zu bearbeiten,
    - und gegebenenfalls erneut abgeben.
- **[unerwartet]**
  - **Abgegeben, aber gemäß der Vorgabe nicht erwartet:**
    - Es kann zwar sein, dass du diese Dateien gar nicht explizit abgeben musstest,
      - aber je nach Aufgabenstellung können sie trotzdem erwüscht sein.
- **[fehlt]**
  - **Nicht abgegeben, aber gemäß der Vorgabe eigentlich erwartet:**
    - Diese Dateien hättest du eigentlich auch abgeben müssen.
    - Falls dies nicht deine Absicht war, solltest du kontrollieren,
      - ob du diese Dateien vergessen hast zu Bearbeiten,
      - oder ob du diese Dateien nicht zum Zip-Archiv hinzugefügt hast,
      - oder ob du sie vielleicht auch einfach nur falsch benannt hast,
    - und gegebenenfalls erneut abgeben.

Abbildung 4.7: Abgabefeedback

Das Feedback aus Abbildung 4.7 besagt Folgendes: Zwar scheint der Student die Aufgabe 1 abgegeben zu haben, allerdings hat er die Teilaufgaben a) und b) wohl nicht korrekt abgegeben. Möglicherweise hat er aber lediglich vergessen die entsprechenden Unterverzeichnisse anzugeben, in welche die Dateien **blue.png** und **text.txt** vielleicht gehören könnten. Aufgabe 2 hat er entweder nicht bearbeitet, oder die geforderte **TimeStopper.jar** nicht richtig erzeugt, denn diese Datei scheint leer zu sein. Wie man sieht, wurde die Abgabe des Studenten trotzdem akzeptiert, um einen reibungslosen Ablauf zu gewährleisten. Allerdings hat der Student jetzt durch das Feedback die Möglichkeit bis zur Abgabefrist eine verbesserte Version seiner Lösung abzugeben.

Nach dem Ablauf der Abgabefrist teilt der Übungsleiter die Abgaben den Korrektoren zu.

Lösungen verteilen (1 von 2)

InfoSys - 123 Blatt: Blatt 11 ->

Verteilen nach Anzahl | Verteilen nach Anteilen | Korrekturanteile festlegen

Die Validierung von 22 Abgaben gegen Ihre Vorgabe steht noch aus.
(3 Abgaben wurden bereits validiert.)

Sie sollten die Lösungen erst danach verteilen,
damit die Tutoren die Ergebnisse der Validierung berücksichtigen können.

Sobald die Validierung abgeschlossen ist, erhalten Sie automatisch eine Email!

Zu verteilende Lösungen: 25

Tutoren	Lösungen
Heinz Schmidt	25

verteilen

Abbildung 4.8: Ausstehende Validierung

Die Funktionstests für die Abgaben werden auf dem Validierungsserver ausgeführt, um den Betrieb von UniWorX nicht zu stören. Solange diese Tests noch nicht abgeschlossen sind, bekommt der Übungsleiter eine Meldung, wie in Abbildung 4.8 zu sehen: Hier sind von 25 Abgaben erst 3 Abgaben validiert, die Validierung der restlichen 22 Abgaben steht noch aus. Wie der Validierungsserver an die Abgaben kommt, und wie seine Kommunikation mit UniWorX konkret aussieht, wird später noch genauer erläutert.

Was aber auch hier wieder deutlich wird, ist ein allgemeines Prinzip, das der Integration mit UniWorX zugrunde gelegt wurde: Die Integration sollte so unaufdringlich wie möglich sein. Der Übungsleiter kann also ebenso die Verteilung vornehmen obwohl die Validierung noch nicht abgeschlossen ist, wie der Student abgeben kann, obwohl seine Abgabe nicht konform zu der Spezifikation ist. Und auch der Validierungsserver ist aus UniWorX ausgelagert, obwohl eine direkte Integration in UniWorX die Umsetzung erleichtert hätte. Der Grund hierfür ist folgender: Ein Abgabesystem ist ein kritisches System, das stets reibungslos funktionieren muss. Ein Zusammenbruch aus Performance-Gründen ist da ebenso unverzeihlich, wie eine Abgabe, die nicht möglich ist, weil eine nicht mit der Aufgabenstellung übereinstimmende Spezifikation dafür sorgt, dass alle Abgaben abgelehnt werden. Auch ein fehlerhafter oder einfach nur äußerst langwieriger Funktionstest, wegen dem die Validierung verzögert wird, darf nicht dafür sorgen, dass die Abgaben nur mit großer Verspätung korrigiert werden können. So können alle Zusatzfunktionen ihren Zweck erfüllen, ohne dadurch die Zuverlässigkeit oder die Verfügbarkeit des Abgabesystems zu gefährden.

4.4 Validierungsserver

Der Validierungsserver ist eine Standalone-Java-Anwendung, die asynchron mit UniWorX über Netzlaufwerke kommuniziert. Bevor aber nun die Funktionsweise des Validierungsservers im Detail aufgezeigt wird, soll zuerst dessen Zusammenspiel und Kommunikation mit UniWorX betrachtet werden.

4.4.1 Zusammenspiel mit UniWorX

Grundsätzlich war das Hauptziel das Zusammenspiel so zu gestalten, dass UniWorX und Validierungsserver soweit wie möglich voneinander entkoppelt werden. Beide Server sollen unabhängig voneinander laufen, und nicht in ihrer Funktionsweise eingeschränkt werden, auch falls der jeweils andere Server kurz- oder mittelfristig nicht verfügbar sein sollte.

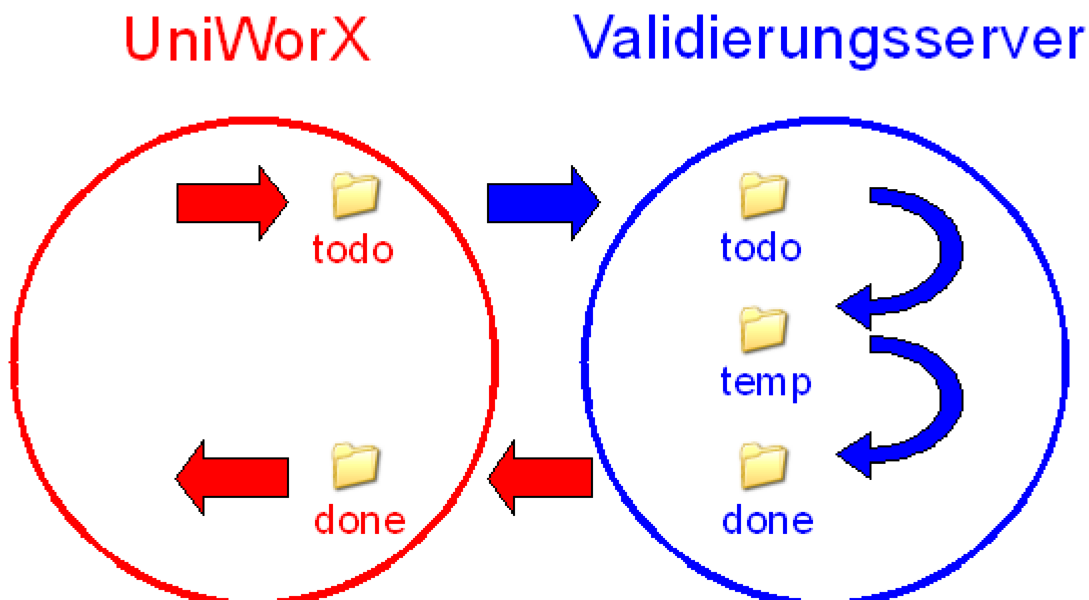


Abbildung 4.9: Zusammenspiel von UniWorX und Validierungsserver

Wie Abbildung 4.9 zeigt, läuft die Kommunikation zwischen UniWorX und dem Validierungsserver daher indirekt ab: Über konfigurierbare Verzeichnisse werden die Abgaben ausgetauscht. Zu validierende Abgaben werden von UniWorX zusammen mit ihrer Spezifikation in ein todo-Verzeichnis abgelegt. Von dort holt der Validierungsserver die Abgaben ab, indem er sie in sein eigenes todo-Verzeichnis verschiebt. Die Validierung selbst wird dann auf einer Kopie im temp-Verzeichnis ausgeführt, und sobald diese abgeschlossen ist, wird das Ergebnis im done-Verzeichnis abgelegt.

Von dort aus holt UniWorX schließlich das Ergebnis ab, und verschiebt es zu sich in das eigene done-Verzeichnis. Danach wird das Validierungsergebnis in UniWorX archiviert, und die entsprechende Abgabe als validiert markiert. Hierbei entspricht jede Abgabe einem eigenen Unterverzeichnis, dessen Name der eindeutigen UniWorX-ID der Abgabe entspricht. Diese ID wird über den gesamten Validierungsprozess über beibehalten, und erlaubt es UniWorX so das Validierungsergebnis wieder korrekt zuzuordnen.

4.4.2 Kommunikation mit UniWorX

Da prinzipiell sowohl UniWorX als auch der Validierungsserver jederzeit angehalten werden können, oder ein Neustart erforderlich werden kann, muss dies bei jedem Kopiervorgang zwischen UniWorX und Validierungsserver berücksichtigt werden. Ein Kopiervorgang folgt daher immer folgendem Protokoll:

1. Dateien vom Quellverzeichnis ins Zielverzeichnis kopieren
2. Zielverzeichnis auf Vollständigkeit und Korrektheit überprüfen
3. Markerdatei im Zielverzeichnis erstellen
4. Dateien im Quellverzeichnis löschen

Bei jedem Schritt im Validierungsprozess wird daher am Anfang stets überprüft, ob die Markerdatei schon vorhanden ist, sonst wird die entsprechende Abgabe einfach ignoriert. Dadurch gehen nach einem Absturz oder Neustart höchstens noch nicht fertige Validierungsergebnisse verloren – was aber nichts macht, da dann die Abgaben erneut validiert werden, weil diese erst aus dem todo-Verzeichnis des Validierungsserver gelöscht werden, sobald die Validierung vollständig abgeschlossen ist.

Sollte also UniWorX oder der Validierungsserver abstürzen, können beide problemlos ihre Arbeit wieder aufnehmen. Auch kann UniWorX problemlos weiterlaufen, solange der Validierungsserver nicht verfügbar ist. In diesem Fall würden die zu validierenden Abgaben im todo-Verzeichnis abgelegt, bis sie irgendwann vom Validierungsserver wieder abgeholt werden. Auch könnten bei einem längeren Ausfall die Abgaben trotzdem auf die Korrektoren verteilt werden, sofern auf die Validierungsergebnisse verzichtet wird. Im umgekehrten Fall, wenn UniWorX ausfallen sollte, während der Validierungsserver weiterlaufen würde, würde dieser alle Abgaben, die er bereits abgeholt hat, abarbeiten, und die resultierenden Validierungsergebnisse im done-Verzeichnis ablegen. Nach dem Neustart von UniWorX würde dieses die fertigen Validierungsergebnisse abholen, und gleichzeitig wieder damit beginnen, neue zu validierende Abgaben für den Validierungsserver bereitzustellen.

Wie man sieht, wurde die Entkoppelung von UniWorX und Validierungsserver so gestaltet, dass beide Server unabhängig voneinander laufen können, und der Betrieb auch bei dem Ausfall von beiden oder nur einem Server problemlos wiederaufgenommen werden kann. Dies konnte durch die beschriebene Kommunikation erreicht werden, die nur indirekt über bestimmte Verzeichnisse und Dateien erfolgt, weswegen sie komplett asynchron ablaufen muss. Diese Einschränkung würde es vermutlich etwas umständlich gestalten, wollte man dem Studenten direkt nach seiner Abgabe auch die Ergebnisse der funktionalen Tests präsentieren. Da die Abgaben sich meist kurz vor der Abgabefrist häufen, und die Auswertung der Funktionstest ja durchaus länger dauern kann, wäre dies, selbst falls es möglich wäre, nicht unbedingt sinnvoll. Insofern wurde hier eine klare Entscheidung zu Gunsten der Robustheit und Verfügbarkeit des Systems getroffen, deren Vorteile die Nachteile deutlich überwiegen.

Konkret realisiert werden die Kopiervorgänge zwischen UniWorX und Validierungsserver durch die in Abbildung 4.10 zu sehenden zwei FileMover-Threads, die beide gemäß dem beschriebenen Protokoll operieren.

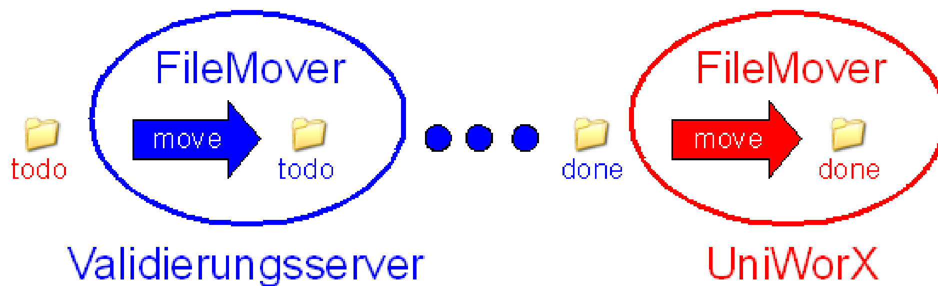


Abbildung 4.10: FileMover-Threads

4.4.3 Validierungsprozess

Neben dem FileMover-Thread läuft im Validierungsservers eine dynamische Anzahl an Threads, welche unabhängig voneinander verschiedene Teilfunktionen des Validierungsprozesses übernehmen.

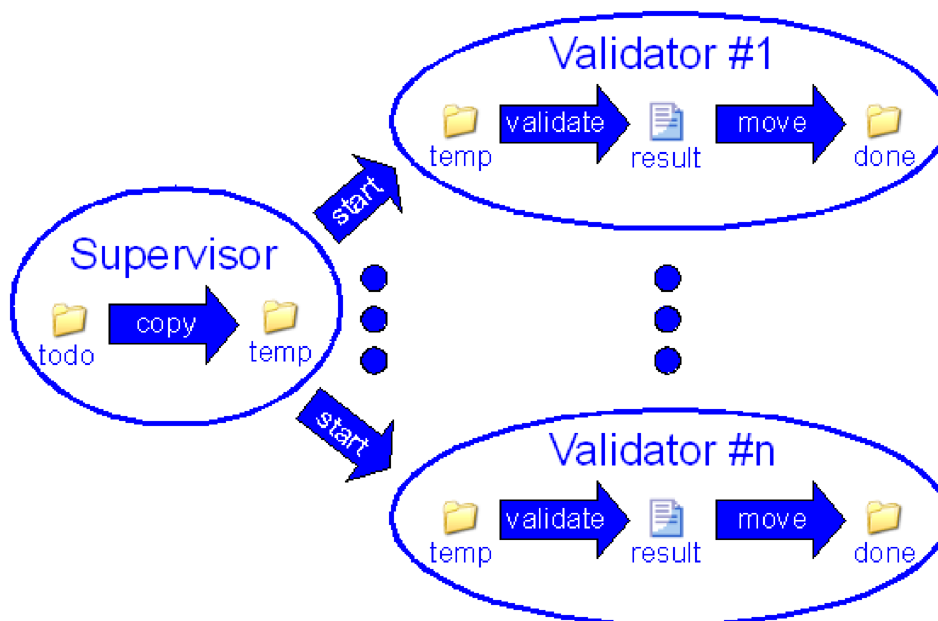


Abbildung 4.11: Supervisor-Thread und Validator-Threads

In Abbildung 4.11 kann man den Supervisor-Thread sehen, der, sobald er eine abgeholte Abgabe entdeckt, diese ins temp-Verzeichnis kopiert, und einen dazugehörigen Validator-Thread startet, bis zu einer definierten Obergrenze an Threads. Sobald nun ein Validator-Thread die Validierung seiner Abgabe fertiggestellt hat, stellt er das Ergebnis im done-Verzeichnis bereit, löscht die Abgabe aus dem todo-Verzeichnis, und terminiert dann.

Jeder Validator-Thread nimmt die erwähnte Validierung nach dem Schema vor, das in Abbildung 4.12 abgebildet ist:

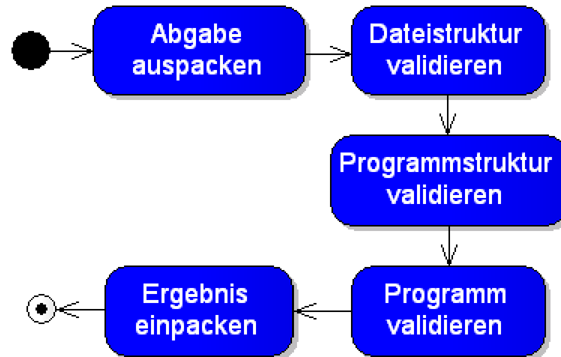


Abbildung 4.12: Validierungsprozess

Da sowohl Abgabe als auch die beigelegte Spezifikation als Zip-Archiv vorliegen, müssen beide am Anfang entpackt werden. Ebenso wird am Ende das Ergebnis wieder in ein Zip-Archiv verpackt. Bei der Validierung wird zuerst die Dateistruktur, danach die Programmstruktur, und schließlich das Programm selbst validiert. Die Validierungen bauen logisch aufeinander auf, denn nicht vorhandene Dateien brauchen nicht auf ihre Programmstruktur getestet werden, und Programme, deren Struktur nicht der Spezifikation entspricht, können eventuell auch nicht gegen die Funktionstests validiert werden.

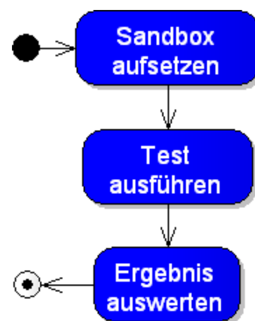


Abbildung 4.13: Testvorgang

In Abbildung 4.13 kann man den Testvorgang sehen, der bei der Validierung des Programmes für jeden Funktionstest durchgeführt wird: Zuerst wird eine Sandbox vorbereitet, welche exakt die benötigten Sicherheitsprivilegien zur Verfügung stellt, die spezifiziert wurden. Die Sandbox wird dann als separater Prozess gestartet, mit der Anweisung den Test auszuführen und das Testergebnis zu speichern. Bevor im Validator-Thread auf das Testergebnis gewartet wird, wird ein Wächter-Thread angestoßen. Terminiert der Test in der Sandbox nicht nach der spezifizierten maximalen Laufzeit, wird der Prozess der Sandbox vom Wächter-Thread terminiert, und der Validator-Thread vermerkt ein leeres Testergebnis.

5 Ergebnisse und Ausblick

Nachdem wir nun auch den Validierungsserver gesehen haben, haben wir damit das gesamte Design von UniBene gesehen. Daher sollen nun die Ergebnisse diskutiert werden, bevor noch ein Ausblick auf künftige Erweiterungsmöglichkeiten, sowie ein abschließendes Fazit gegeben werden soll.

5.1 Methodik

Als Erstes stellt sich natürlich die Frage, ob sich die gewählte Methodik im Laufe der Entwicklung bewährt hat: Der iterativ-inkrementelle Entwicklungsprozess war eindeutig eine große Hilfe bei der Integration, da so das gewählte Design schon sehr früh auf seine Tauglichkeit hin überprüft werden konnte. Auch konnte das Design durch das frühe Feedback mit jeder Iteration weiter verfeinert und verbessert werden, sodass in der Implementierung effektiv einige Arbeit eingespart werden konnte. Was die Projektplanung betrifft, so muss man sagen, dass die ursprünglichen Schätzungen im Großen und Ganzen ein guter Indikator für die tatsächlichen Aufwände waren, auch wenn es im Einzelfall natürlich Abweichungen gab. Dadurch stand stets ein gutes Maß für den Projektfortschritt zur Verfügung, was es ermöglichte eine sinnvolle und robuste Planung durchzuhalten.

5.2 Anforderungen

Als Folge der nach Plan verlaufenden Entwicklung war es auch möglich, die effiziente Erfüllung der anfangs formulierten Anforderungen zu gewährleisten. Der Abgabeprozess konnte komplett implementiert werden: Von dem Erstellen der Spezifikation, über das Hochladen in UniWorX, das Feedback bei der Abgabe in UniWorX, bis zur Validierung der Abgaben sowohl strukturell, als auch funktional. Aber auch die anderen Anforderungen konnten berücksichtigt werden: Die Integration in UniWorX konnte erfolgen, ohne die Verfügbarkeit und die Sicherheit von UniWorX einzuschränken; die Sicherheit bei der Ausführung der Funktionstests konnte durch Sandboxing gewährleistet werden; und auch die geforderte Sprachunabhängigkeit konnte beim Design und der Umsetzung, wie geplant, berücksichtigt werden.

Zu der Sprachunabhängigkeit muss noch erwähnt werden, dass obwohl diese im Design berücksichtigt wurde, bisher nur Validierungen für die Programmiersprache Java umgesetzt wurden. So ist das Spezifikationsformat leicht erweiterbar, und auch bei der funktionalen Validierung können problemlos weitere Module eingehängt werden. Bei der Umsetzung dieser Module muss man aber eventuell das, was einem durch das Sicherheitsmodell in Java einfach gemacht wird, in einer anderen Programmiersprache selbst bereitstellen.

5.3 Erweiterungsmöglichkeiten

Über die umgesetzte Funktionalität hinaus, gibt es zahlreiche Erweiterungsmöglichkeiten. Einige, die bereits für das Bonus-Release angedacht wurden, sollen hier kurz vorgestellt werden.

5.3.1 Spezifikationseditor

Angefangen beim Spezifikationseditor: Dieser könnte enger mit UniWorX verzahnt werden, so dass Spezifikationen nach ihrer Erstellung automatisch in UniWorX verfügbar würden, und Spezifikationen für bestehende Übungsblätter im Editor einfach aus UniWorX geladen werden könnten. Auch könnte man eventuell überlegen, den Editor zusätzlich als Webanwendung zur Verfügung zu stellen, um direkt in UniWorX Spezifikationen erstellen und ändern zu können. Insbesondere falls man nur einfache strukturelle Vorgaben machen wollte, müsste man dafür nicht extra ein lokales Programm starten, was den initialen Aufwand für die Aufgabensteller senken würde.

5.3.2 Integration in UniWorX

Ebenfalls könnte die Integration in UniWorX erweitert werden: Momentan bekommen die Studenten ihre funktionalen Validierungsergebnisse erst mit der Korrektur zu sehen. Denkbar wäre, diese bereits vor der Korrektur in UniWorX als Feedback verfügbar zu machen. Auch könnte man den Studenten zusammen mit der existierenden Vorlage zusätzlich noch die spezifizierten Unit-Tests zur Verfügung stellen. Dann hätten die Studenten diese schon bei der Bearbeitung als Hilfestellung zur Verfügung.

5.3.3 Client für die Studenten

Man könnte sogar noch einen Schritt weitergehen, und nicht nur die Tests zum manuellen Ausführen zur Verfügung stellen. Stattdessen könnte man einen Client für die Studenten entwickeln, der dann für die Studenten Folgendes leisten könnte: Von dem Anlegen einer Vorlage für ein Übungsblatt, über die strukturelle Überprüfung, bis hin zum automatischen Ausführen der Unit-Tests, wäre hier Vieles denkbar, was den Studenten insbesondere durch das unmittelbare Feedback dabei helfen könnte, bessere Lösungen abzugeben.

5.3.4 Client für die Korrektoren

Ein ähnlicher Client wäre auch für die Korrektoren denkbar, der eine Liste aller Abgaben anzeigt, und die von den einzelnen Studenten jeweils bearbeiteten Aufgaben kenntlich macht. Zudem könnte er die direkte Bearbeitung der Lösungen zur Korrektur anbieten, sowie im Anschluss vermerken, welche bereits korrigiert sind. Darüber hinaus könnten zu den einzelnen Aufgaben vom Aufgabensteller definierte Checklisten zur Korrektur angezeigt werden, beispielsweise um die Bepunktung festzulegen. Dies könnte die Korrektoren dabei unterstützen, die Abgaben noch systematischer zu korrigieren.

5.3.5 Andere Erweiterungsmöglichkeiten

Über die genannten Szenarien hinaus, gibt es sicher noch andere Erweiterungsmöglichkeiten. Doch erscheint es sinnvoll, erst den produktiven Einsatz abzuwarten, um praktische Erfahrungen mit UniBene zu sammeln. Sowohl die Relevanz der genannten Möglichkeiten, als auch der Bedarf für andere Erweiterungen wird dadurch voraussichtlich leichter ersichtlich werden.

5.4 Fazit

Im produktiven Einsatz wird man darüber hinaus beobachten können, inwiefern Abgabe- und Korrekturprozess durch UniBene verbessert werden. Doch sollte man keine Wunder erwarten – einen Korrektor ersetzen kann man damit nicht. Aber dafür war UniBene ja auch nicht gedacht. Das Ziel war vielmehr, den Studenten, insbesondere durch das frühe Feedback, unnötige und ärgerliche Fehler zu ersparen, die sie noch hätten ausbessern können. Dem Aufgabensteller wird außerdem die Möglichkeit gegeben, seine Aufgaben nicht nur präzise zu spezifizieren, sondern auch die Gewissheit zu haben, dass sowohl Studenten als auch Korrektoren diese Spezifikation einfach nachvollziehen können. Dem Korrektor soll Arbeit abgenommen werden durch die automatisierten strukturellen und funktionalen Tests, deren Ergebnisse ihm zu jeder Abgabe zur Verfügung gestellt werden, damit er dank der neu gewonnenen Zeit die Abgaben auch über ihre bloße funktionale Korrektheit hinaus bewerten kann. Dies wiederum soll die Korrektur für den Studenten verbessern. Eine Korrektur zu bekommen, die sich dank neu gewonnener Zeit nicht nur auf ein richtig oder falsch beschränken muss, ist insbesondere für Studenten mit geringer Programmiererfahrung besonders hilfreich. Aber auch Studenten mit mehr Programmiererfahrung können von Anregungen und Tipps profitieren, welche die Art und Weise kommentieren, wie das abgegebene Programm umgesetzt wurde. Gilt es doch den Studenten folgende Botschaft mitzugeben: Schreibe nicht nur die richtigen Programme, sondern schreibe deine Programme auch richtig! Denn wie heißt es so schön bei Martin Fowler:

> „Any fool can write code that a computer can understand.
> Good programmers write code that humans can understand." [FBB+99]

Wenn also UniBene einen kleinen Beitrag dazu leisten kann, dass die Aufgabensteller ihre Aufgaben exakter spezifizieren können, dass die Korrektoren die selben Abgaben in weniger Zeit profunder korrigieren können, und dass die Studenten von einem größeren Lerneffekt profitieren können, dann und nur dann hat diese Arbeit ihren Sinn und Zweck auch erfüllt.

Literatur

[BD03] Jason Brittain and Ian F. Darwin. *Tomcat: The Definitive Guide.* O'Reilly, 2003.

[BKEdN08] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. *YAML 1.2 Specification.* `http://yaml.org/spec/1.2/`, 2008.

[Cav04] Chuck Cavaness. *Programming Jakarta Struts.* O'Reilly, second edition, 2004.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[FF04] Ira R. Forman and Nate Forman. *Java Reflection in Action.* Manning Publications, 2004.

[GED03] Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation.* Prentice Hall PTR, second edition, 2003.